Building Strongly-Consistent Systems Resilient to Failures, Partitions, and Slowdowns

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the registrar's office.

Thèse n. 8595 2022
présentée le 30 novembre 2022
à la Faculté informatique et communications
Laboratoire de systèmes décentralisés et distribués
Programme doctoral en informatique et communications
École polytechnique fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Cristina Băsescu

acceptée sur proposition du jury :

Prof R. Guerraoui, président du jury
Prof B. A. Ford, directeur de thèse
Prof I. Stoica, rapporteur
Prof R. van Renesse, rapporteur
Prof K. Argyraki, rapporteuse

Lausanne, EPFL, 2022

“If I have seen further, it is by standing on the shoulders of giants.”
—Isaac Newton

To Elena, Constantin, Henry, always unconditionally supportive.

Acknowledgements

I would like to thank my advisor, Prof. Bryan Ford for believing in me, and for spending his time and energy to guide and support me over the years. Bryan is unafraid to tackle the biggest research problems; he inspires by example, and his optimism and courage are contagious. It has been an immense pleasure to work with him. Bryan is also a kind and patient person, and always manages to find time, somehow. I am immensely indebted to him.

I am extremely grateful to my thesis committee members, Prof. Katerina Argyraki, Prof. Robbert van Renesse and Prof. Ion Stoica, for serving on my committee and for taking the time to review my thesis, and Prof. Rachid Guerraoui, for presiding over my thesis committee.

My time in DEDIS would not have been the same without the amazing researchers and engineers, with whom I shared numerous brainstorming sessions, coffees, lunches and dinners. Cey, David F., David L., Gaylor, Georgia, Haoqian, Henry, Jean, Jeff, João, Kelong, Kirill, Lefteris, Linus, Louis-Henri, Ludovic, Nicolas, Noémien, Pasindu, Philipp, Pierluca, Simone, Stevens and Vero.

Many thanks go to my collaborators: Georgia, Cey, Haoqian, Pasindu, Lefteris, Kirill, Kelong, Gaylor, Fitz, Jose, Pierluca and Vero, for their hard work and feedback through all the deadlines. Your support was invaluable.

Thank you to the administrative staff: Sandra, Patricia and Maggy, and to our editing wizard Holly. They are always very effective and help immensely.

I would like to thank my friends: Andreea V., Lăcră, Cristina, Vali, Simona, Roxana, Andreea W., Dana for all the fun times over the years.

I am very grateful to my husband's family for their encouragement and support; they even found the time to read drafts of my papers. I keep close to my heart many of the moments we spend together.

Special thanks go to my parents, Elena and Constantin, and to my husband Henry, to whom this thesis is dedicated. They always find the time to listen and understand me, and support me in an infinity of ways. They never once complain when I am such a pain. Thank you for your love.

Lausanne, December 30, 2022 C. B.

Abstract

Distributed systems designers typically strive to improve performance and preserve availability despite failures or attacks; but, when strong consistency is also needed, they encounter fundamental limitations. The bottleneck is in replica coordination, which is impacted by partitions and slowdowns that can occur anywhere. We believe the present ecosystem fails to recognize that not all failures and partitions are supposed to be equal — at least from a user-centric performance and availability standpoint. Failures distant from a user should intuitively be less likely to affect that user. Today's ecosystem fails this test, however, despite high-availability best practices. For example, correlated and cascading failures, caused by misconfiguration, bugs, and network partitions, often invalidate the cloud's assumptions of failure independence. Likewise, large-scale or accurately targeted routing or denial of service attacks can slow or halt a distributed ledger or compromise its security.

We believe that distributed systems designers and practitioners can and should build reliable, responsive systems by making Lamport exposure and asynchrony central design considerations. We propose that distributed services need not and should not expose local activities to distant failures or partitions, no matter how severe. Limix is the first exposure-limiting metadata configuration service that addresses this problem. Limix insulates global strongly-consistent data-plane services and objects from remote failures and partitions by ensuring that the definitive, strongly-consistent metadata for every object is always confined to the same zone as the object itself. Nyle is a trust-but-verify distributed ledger architecture that limits transaction exposure. While employing similar design principles as Limix, Nyle additionally supports an environment with Byzantine nodes and potentially compromised regions with an elevated presence of attackers, and enables asymmetric user trust preferences. Both Limix and Nyle outperform related work in terms of availability, at a manageable overhead. We also demonstrate, through the design of QSC, that consensus protocols can deal with network asynchrony without relying on common coins, having the potential to make consensus more responsive and more practical.

Keywords. Resilience, failures, partitions, strong consistency, Lamport exposure, responsiveness, metadata configuration service, blockchain, crash-fault tolerant asynchronous consensus

Résumé

Les concepteurs de systèmes distribués cherchent généralement à améliorer les performances des systèmes et à préserver leur disponibilité malgré les défaillances ou les attaques ; mais lorsqu'une cohérence forte est également nécessaire, cette démarche se heurte à des limites fondamentales. Le problème se situe au niveau de la coordination de la réplication des données, qui est affectée par les ralentissements et les partitions du réseau qui peuvent survenir n'importe où. Nous pensons que l'écosystème actuel ne reconnaît pas que toutes les défaillances et partitions ne sont pas censées être égales – du moins du point de vue des performances et de la disponibilité perçues par l'utilisateur. Les défaillances éloignées d'un utilisateur devraient intuitivement être moins susceptibles d'affecter cet utilisateur. L'écosystème actuel n'atteint toutefois pas cet objectif, malgré l'application des bonnes pratiques de haute disponibilité. Par exemple, les défaillances corrélées ou en cascade, causées par une mauvaise configuration, des bugs ou des partitions de réseau, invalident souvent les hypothèses d'indépendance des défaillances du cloud. De même, des attaques sur le routage ou de déni de service – de grande ampleur ou minutieusement ciblées – peuvent ralentir ou arrêter une blockchain ou compromettre sa sécurité.

Nous pensons que les concepteurs et experts en systèmes distribués peuvent et doivent construire des systèmes fiables et réactifs en considérant, dès la phase de conception, l'exposition de Lamport et l'asynchronie comme des éléments essentiels de tels systèmes. En ce sens, nous pensons que les services distribués n'ont pas besoin, et ne doivent pas, exposer leurs activités locales à des défaillances ou à des partitions de réseau distantes, quelles que soient leurs gravités. Limix est le premier service de configuration de métadonnées qui traite ce problème et limite l'exposition des systèmes distribués. Limix isole les services et les données globales de systèmes à cohérence forte des défaillances et partitions distantes en garantissant que les métadonnées définitives et fortement cohérentes de chaque objet sont toujours confinées dans la même zone que l'objet lui-même. Nyle est une architecture de blockchain de type "fais confiance, mais vérifie" qui limite l'exposition des transactions. Tout en utilisant des principes de conception similaires à ceux de Limix, Nyle prend en compte un environnement avec des nœuds Byzantins et des régions potentiellement compromises par une présence élevée d'attaquants, et permet aux utilisateurs d'exprimer des préférences de confiance asymétriques. Limix et Nyle surpassent les systèmes similaires en termes de disponibilité et leur déploiement n'entraîne qu'un modeste surcoût. Nous démontrons également, par la conception de QSC, que les protocoles de consensus peuvent traiter l'asynchronie du réseau sans s'appuyer sur des common coins, ce qui a le potentiel de rendre le consensus plus réactif et plus pratique.

Résumé

Mots-clés. Résilience, défaillances, partitions, cohérence forte, réactivité, exposition de Lamport, service de configuration de métadonnées, blockchain, consensus asynchrone tolérant les pannes franches

Contents

1 Introduction

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> Leslie Lamport

This chapter introduces the problem that this thesis addresses: the availability and performance of strongly consistent distributed systems. It establishes the context for geo-replicated cloud services, distributed ledgers and asynchronous consensus, and identifies opportunities for achieving immunity from remote failures, partitions and slowdowns. Finally, it highlights the contributions of the thesis and presents the thesis structure.

1.1 The Problem

When building distributed systems, we usually aspire to maximize performance and maintain availability despite failures or attacks. In many cases, we would also prefer strong consistency, which makes it even more challenging to maintain performance. Strong consistency requires coordination among replicas in systems using standard availability techniques, such as replication. Replica coordination may impair performance. Especially when systems span a wide area, coordination increases response latency and unavailability, because partitions and slowdowns occurring anywhere disturb coordination. The harsh compromise presents itself as the CAP theorem [57, 58], which in theory, restricts our guarantees to a choice of only two properties among consistency, availability, and partition-tolerance.

Several systems - such as cloud services and distributed ledgers - attempt to simplify the CAP choice. Cloud platforms strive to reduce network partitions to "rare events": assuming partitions are sufficiently unlikely, we can achieve strong consistency with high availability. In practice, however, individually-rare but correlated or cascading failures frequently puncture

the cloud's illusion of having "solved" the CAP tradeoff. Leslie Lamport's quote above, while dating from the 80s, pointedly describes today's state of affairs. Reality abounds with unavailability and slowdowns, caused by misconfiguration [2, 68], cascading failures due to partially functional equipment [81], gray failures [66], and partitions due to software bugs [22, 61], despite best-practice geo-replication. A single Fastly [50] customer recently triggered a bug causing a global outage [65]. Failures such as these result in widespread downtime and loss of revenue [13].

Likewise, distributed ledgers attempt to eliminate the concern of partitions. Some ledgers prioritize availability and simply assume for consistency that partitions are unlikely to last "too long" [49, 92, 128]. More recent designs prioritize consistency and assume that a sufficient number of validators are alive and reachable [75, 76] or select validators through uptime measurements to increase the network's reliability [19]. Yet, reality does not conform to these assumptions. Large-scale or carefully targeted routing attacks can partition the network by redirecting traffic through an attacker-controlled autonomous systems (AS) that drops it [14], denial-of-service attacks can cut communication by exploiting network sharing effects [109, 111]. Securing Internet routing can counter some of these attacks [46, 132], but progress is slow and these are not deployed in practice because they require agreement among stakeholders [112].

We ask a simple question: is it truly acceptable that "the failure of a computer you didn't even know existed" – located anywhere in the world and run by one of many companies you also probably didn't even know existed – "can render your own computer unusable"? Is it fair or responsible that we *expose* users continually to a vast multitude of accidental or malicious failures and slowdown risks that are both *non-transparent* (infrastructure and operators "you didn't know existed") and *unbounded* (potentially located anywhere) – no matter how *simple or localized* a user's actual computing needs might be? And what would the implications be for distributed service designs if we were to decide that the answer to this question is *no*?

Immunity to remote failures and partitions. Suppose Alice is collaborating with Bob, another user located in the same city or corporate campus, using a team messaging and collaboration application such as Slack [106]. Or Alice may purchase a coffee from Bob, a merchant, using tokens stored on a blockchain. We conceptually define the *Lamport exposure* of Alice's activity as the set of infrastructure elements anywhere that could contribute to halting her work. That is, Alice's Lamport exposure is the complete set of infrastructure failure risks – including cloud services, network, power, etc. – that her particular activities depends on, directly or indirectly, and whose individual or joint failure could ultimately grind her work to a halt.

We believe distributed systems should be designed to make Lamport exposure *transparent* to users, under the *control* of users, and ideally *limited* to the smallest set of failure risks feasible for a given application. As a purely-illustrative example, simple packet forwarding, in an idealized network run by a shortest-path routing algorithm, has a natural *Lamport exposure-limiting* property. If Alice is communicating with Bob at a network distance of Δ,

then *no* combination of node or link failures beyond a distance of Δ from Alice can halt Alice's activity – because nodes and links beyond this radius cannot be on the shortest path. Alice's chat with Bob is thus "immune" to failures farther than Δ from her, hence her chat's Lamport exposure is *limited* to a radius of Δ.

The real Internet fails to guarantee such exposure limiting even for basic TCP/IP communication, due to congestion or complex off-path failures such as BGP hijacking, for example. And almost no complex applications or services higher up the stack – especially those requiring strong consistency – currently guarantee any readily-definable exposure limits. If Alice's activity A depends on strongly-consistent state replicated globally, then A's Lamport exposure is generally a large, global set of devices that Alice probably "didn't even know existed" – so A is vulnerable to correlated failures and partitions arising anywhere even if they are uncommon.

But the most critical *needs* for access are often localized: Alice cares most that her data and services are accessible from where she usually accesses them. Trust is also localized: people tend to trust local businesses [4] and governments [6] more and may similarly be more willing to trust digital service providers if they can show that critical services depend only on local resources. And payments have a strong locality property with respect to geography: in 2020, in certain regions, there were at least twice as many intra-border payments than cross-border payments [41]. Thus, limiting a service's Lamport exposure, if achievable, could translate into availability guarantees more meaningful to users.

Resilience to slowdowns. Achieving immunity from remote failures and partitions, e.g., by limiting a service's Lamport exposure, is a first necessary step to resilient systems. However, an available service does not imply and is not sufficient for a responsive service. Even with consensus groups within localities, services can still be exposed to *intra-locality* failures and network hiccups, i.e., network asynchrony, caused by partially functional equipment or (temporary) misconfiguration, or even with malicious intent by an adversary that applies selective censorship on network links. Such asynchrony may cause unnecessary delays in consensus protocols usually used for coordination that rely on timing assumptions. We loosely borrow the responsiveness definition of Yin et al [131] as the property of an algorithm to drive consensus "at the speed of the actual (vs maximum) network delay". Synchronous and partially/weakly synchronous deterministic consensus protocols are not responsive when the network is asynchronous [89] - in fact, they cannot guarantee progress [51]. Asynchronous protocols can probabilistically guarantee progress, but are rarely, if ever, deployed in practice.

We believe practitioners would prefer to deploy asynchronous consensus protocols, because they operate as robustly as possible, in terms of both availability and performance, in the face of arbitrarily adverse network conditions, and not just in "normal-case" conditions – granted that asynchronous consensus was practical. Towards this goal, we are interested in reducing the constraints behind asynchronous protocols, because they may hinder applicability. In particular, existing asynchronous consensus require a trusted setup in the form of a common coin [9, 73, 83, 89, 114]. The assumption is more problematic in Byzantine environments,

because corrupting a trusted dealer breaks the entire protocol, whereas in crash-only environments a trusted setup is mostly seen as an administrative burden, which, however, still needs to be protected. We are interested to explore whether a common coin is actually necessary for asynchronous consensus. For crash-stop environments, we develop a protocol that does not require common coins and is responsive, which hopefully can inspire approaches for a Byzantine fault tolerant asynchronous consensus that does not require expensive distributed key generation protocols, but only cheaper verifiable random functions. Besides not needing common coins, our protocol is responsive and arguably elegant, which we hope will boost the applicability of asynchronous consensus in practice.

Contributions. Failures far away from a user should intuitively be less likely to affect that user, but the current ecosystem completely fails to fulfill this intuition. Can we build services that are *usually* available from anywhere, but which can offer *transparency* about their dependencies, and *guarantee* accessibility from where they are typically used or most needed – even in the face of more distant failures or partitions? Coordination protocols should ideally be responsive to changing network conditions, however the current ecosystem only infrequently employs asynchronous protocols, which are capable of managing such situations. Can we eliminate some of the assumptions that hamper the adoption of asynchronous consensus protocols, thereby making them more applicable? This thesis argues that distributed systems designers and practitioners can and should build reliable, responsive systems by making Lamport exposure and asynchrony a core consideration in their design.

1.2 Context

We review the current context in terms of architectures for distributed systems deployments. With example applications, we explain that typical deployments may create implicit but unnecessary dependencies, leading to a suboptimal Lamport exposure. Finally, for each example, we give some insights of how to design and deploy systems with a low exposure.

1.2.1 Geo-replicated cloud services

It is customary to deploy cloud services on a region level and manage cross-region deployments manually. This trend matched the restricted offering of cloud providers that started with a few disjoint, coarse-grained regions, roughly matching (sub)continents. Several providers now lower their latency to users by offering more than 25 geographic regions and 80 availability zones (AZs), with further expansion plans [12, 59, 88]. Despite this surge, the deployment of cross-region and cross-AZ services remains largely the burden of the programmer, and perhaps for a good reason. Providers merely ensure infrastructure-level independence between AZs, leaving customers to handle application-specific dependencies. The Sky [107], a new cloud architecture spanning multiple cloud providers, enables applications to optimize performance vs cost tradeoffs by choosing the right cloud for the required functionality. We believe that

the Sky enhances the urgency of not only multi-region, but cross-provider deployments with meaningful Lamport exposure guarantees.

Geo-replication is not enough. Best-practice geo-replication across regions cannot alone limit Lamport exposure. Consider a collaborative document-editing application where users in different regions edit the document. Storing the document in one fixed region penalizes far-away users with higher exposure to latency and availability risks. Geo-replicating the document across regions requires coordination for consistency among replicas. Such an application typically masks independent replica failures or partitions via consensus [79, 94], which increases exposure in two ways: (1) Data-plane: Even if the requested document has a nearby replica, the user may be unable to access the item without synchronizing with a quorum of replicas or with the leader, which may be slow or unreachable. (2) Control-plane: Even if enough document replicas are in the user's zone, merely locating the document often depends on global state that may be located anywhere. Location metadata cannot usually migrate along with the data, because the system requires a fixed, known entry point for lookup. Akkio, a recent geo-replicated KV store [13], migrates data but not the location database, for example.

Opportunity. We propose Limix, an *exposure-limiting architecture* that addresses this challenge by removing false or implicit but unnecessary dependencies. In Limix's control plane, a set of potentially-overlapping protection areas or *zones* each runs an independent distributed discovery service. Each zone's discovery service ensures that all users within the zone can locate and access any data-plane object or *item* in the same zone without availability or performance dependencies outside that zone. If the (most recent version of the) document is not in that zone, Limix's lookup automatically proceeds to the next-larger overlapping exposure zone. A data-plane item may still be geo-replicated across multiple state-storage *sites* (e.g., data centers); but we consider an item to be within a zone only if all of its replica sites are in that zone. For example, if a data item is replicated across sites in Germany, France, and Italy, with one replica each, then the item is "in" the EU-West but not "in" Germany. EU-West users are guaranteed to contact a consensus quorum and make progress by only relying on infrastructure and logic in the EU-West region. Lamport exposure metrics could be geographic distance or latency, leveraging the clouds' rare partitions and stable inter-region latencies.

Developing Limix is the main focus of Chapter 2, where we show that Limix enables a globally managed key-value store while protecting local accesses from distant failures. The takeaway message of Chapter 2 is that that systematically limiting Lamport exposure significantly improves availability by as much as 30% on synthetic but realistic workloads.

1.2.2 Distributed ledgers

Unlike clouds, which have private interconnects, decentralized peer-to-peer applications such as distributed ledgers (or blockchains) must handle unpredictable wide-area network delays.

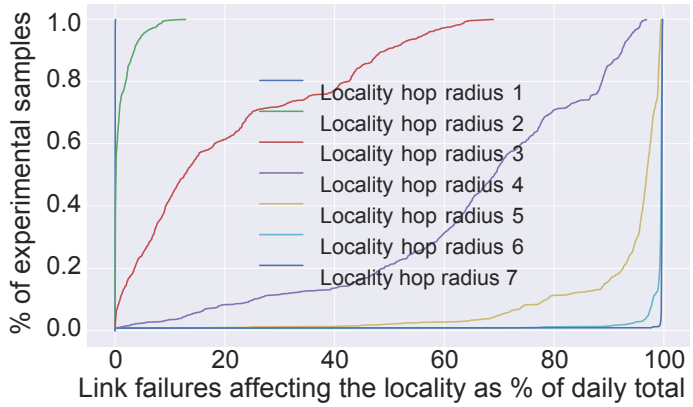


Figure 1.1: CDF of the propagating effect of link failures in Jan.'20 over various exposure localities.

These delays might be due to attacks, e.g., denial of service (DoS) or BGP prefix hijacking, but may also have benign causes, such as autonomous system (AS) misconfiguration or network sharing effects. Such events arise because the Internet is a collaboration between ASes with mutual trust. The BGP routing algorithm continues to operate largely on a trust basis, though more secure proposals such as BGPSec [46] are advancing. Likewise, unrelated traffic shares the Internet links on a best-effort basis, which is why these links are susceptible to DoS attacks [109, 111]. Absent attacks and misconfiguration, Internet routing algorithms have, *in principle*, the Lamport exposure property we aim for. Specifically, these algorithms limit exposure defined through AS routing policies. However, this property gets lost in the upper software layers, as we explain below.

Distributed ledgers over public networks. Consider a public distributed ledger using Byzantine fault tolerant consensus [34] handling token transfers over the Internet. Strong consistency is a must in order to avoid double spending, i.e., the action of spending the same token in multiple transactions. Consider a transaction between a sender Alice in France and a receiver Bob in Germany. Bob needs to wait for a Byzantine quorum of validators to agree on the transaction. In a global blockchain, this quorum likely involves participants outside the borders of France and Germany, increasing the Lamport exposure with all the components and networks necessary for remote communication.

To get a rough, numerical estimate of exposure-limiting opportunity, we make a simplistic computation of the failure exposure of an imaginary exposure-limiting system deployed over the Internet. This illustrative example has many limitations: for example, we would like the experiment to use all types of failures across the software stack, not just presumed link-level failures. We used CAIDA data for the AS graph from January '20, which contains several daily snapshots. We denote as "stable graph" the graph with edges appearing in at least half of these snapshots, and we consider as failures those edges that appear in the stable graph but are missing from a daily snapshot. We run 1000 trials, where each trial builds localities with a

random AS center and hop-count radiuses 1 to 7. For each trial, we select a random day's snapshot and count the number of failed links in the locality compared to the total number of link failures in the snapshot. Figure 1.1 depicts the CDF of the results. Smaller localities are exposed to fewer failures, e.g., users connecting to ISP services within a locality with hop radius 2, in 95% of the cases, are immune to 90% of the failures. Our goal is to have the same level of guarantee for the upper levels of the software stack.

Opportunity. Blockchains may reduce Lamport exposure with a "trust-but-verify" architecture. Such architectures have been proposed in the past, but rely on selecting an unbiased sample of validators [76]. If the selected sample is far away from Alice and Bob, e.g., containing nodes in China, then Lamport exposure remains large. One approach is to define Lamport exposure along legal jurisdictions. The transaction between Alice and Bob might require only enough validators in their two countries to agree. Bob is likely to trust local validators because they are in a common legal framework, and consequences are more readily enforceable. The transaction still propagates globally along increasing exposure boundaries, which independently check the transaction and enable Bob to enforce retroactive verification.

Chapter 3 describes Nyle, a trust-but-verify blockchain architecture that limits transaction exposure. Nyle demonstrates that it is feasible to limit exposure despite attacks, including double-spend attempts by attackers and compromised localities with an increased adversary presence that may equivocate. 60% of transactions have 70% less exposure in Nyle, at the expense of 10x increase in load.

1.2.3 Asynchronous consensus protocols

Asynchronous consensus protocols need to reach a decision despite messages potentially taking an arbitrarily long time to reach their destination. Consensus is a fundamental problem in distributed computing. The theory on asynchronous consensus algorithms dates back to the 1980s, but the field has seen a surge of interest due to consensus applications in wide area networks and distributed ledgers. A consensus algorithm ensures that all replicas of an object make the same decision, which they can use to coordinate on an operation involving that object, for example what is the latest value written for a key in a distributed key-value store, or whether a ledger transaction is valid. Typically, algorithms progress in rounds, with each node making a proposal per round, receiving some proposals, and deciding if possible. In an asynchronous network, a well-known impossibility result states that it is impossible to devise a deterministic algorithm [51]. Thus, to avoid infinitely repeating rounds where no decision is possible, nodes randomize their proposal choice for the next round, in the hope that the new set of proposals would trigger a decision condition.

Common coins are not always practical. For randomization, the majority of protocols rely on an abstraction called (almost) perfect common coin, which can be impractical to implement. Generally speaking, a perfect common coin guarantees that every caller process obtains the

same random output if it gives the same set as input. However, perfect common coins require a trusted dealer for the setup. Entrusting a dealer with the setup in the crash-stop model, where nodes can fail but are not malicious, is generally seen as acceptable, if not mostly an operational burden. For example, the dealer distributes a seed that nodes use to initialize a random number generator, i.e., the coin. Of course, the seed should not leak at any point during the algorithm, otherwise an attacker can easily launch a network attack that subverts randomness. In the Byzantine model, however, because nodes can arbitrarily deviate from the protocol, the trusted dealer that deals cryptographic keys becomes the weak link of the entire protocol. Recent protocols can create a common coin without a trusted dealer, however they are more expensive than consensus itself.

Opportunity. Most recent research has focused on optimizing common coins, as noted above. It is natural to ask, however, whether common coins are necessary for consensus. We design a consensus protocol for the crash fault model that relies only on node-private randomness. The design could pave the way to Byzantine protocols that do not need common coins, and perhaps to more practical consensus. As noted in the introduction, DoS attacks and misconfiguration can create asynchrony, and designs that withstand it would be preferable to those that do not. Asynchronous consensus, when practical, could become the de-facto implementation for coordination protocols.

The focus of Chapter 4 is on QSC, our asynchronous consensus protocol without common coins in a crash fault setting. We demonstrate that it is possible to eliminate the common coins assumption and achieve a practical algorithm, i.e., one that decides within $O(1)$ running time in expectation, which is optimal, and $O(n^3)$ total communication complexity in bits, but only $O(n^2)$ total message complexity. Using our new primitive threshold synchronous broadcast (TSB), we also show that such algorithms may be responsive, e.g., QSC advances at the actual speed of the network. Additionally, TSB helps QSC in reducing design complexity via modularity.

1.3 Contributions and Outline

This section describes the thesis contributions and the thesis scope, summarizes them in the thesis statement, and gives the thesis outline.

1.3.1 Contributions

This thesis makes the following contributions:

1. **Defining and measuring Lamport exposure.** To protect users from distant failures, we must precisely define what “distant” means. We define Lamport exposure and provide several metrics that may be meaningful for different applications: administrative or

legal boundaries, geographic straight-line distance, round-trip time.

2. **Meaningful exposure-limiting zones.** Given an exposure metric, we need to control and limit it systematically, which we achieve through exposure-limiting zones. We base the structure and granularity of zones on the guiding idea of maintaining the locality of interactions. One essential property of zones is being able to meet simultaneous constraints, such as when data protection and sovereignty concerns demand that e.g., data remains local in Germany when accessed locally, and stays in Europe when accessed by European users. A further crucial aspect is providing strong exposure guarantees during object migration, which is useful for systems in which the locations from which objects are accessed change dynamically.

3. **Autozoning.** Even absent explicit contractual or regulatory constraints, ordinary users dislike it when their local activities are brought to a halt by distant outages across the globe. Addressing this common-case challenge, we propose an autozoning scheme that builds on compact graph summarization theory. In brief, a user accessing any data at a distance Δ away is protected from all failures occurring beyond a small multiple of Δ.

4. **Limix: an exposure-limiting metadata service.** We design Limix, the first distributed metadata coordination service that enables global management while protecting local accesses from distant failures. We provide a prototype implementation of Limix on top of the key-value store functionality of CockroachDB. Limix routinely outperforms Physalia [22] by as much as 30% better availability, on synthetic but realistic workloads, as shown by our experiments on AWS.

5. **Nyle: an exposure-limiting distributed ledger.** Nyle is a trust-but-verify distributed ledger architecture. While Nyle follows the same general idea as Limix, the Nyle architecture must tackle two new major challenges: Byzantine nodes, which might deviate arbitrarily from the protocol by e.g., double-spending or by lying about exposure metrics, and potentially compromised zones. Thus, the emphasis is on the security properties, motivating the need for a trust-but-verify architecture, where clients may pick which local zones to trust and afterwards verify that no double spending occurred. Our simulation shows that Nyle significantly reduces exposure compared to Omniledger [76], with 60% of transactions witnessing 70% less exposure in Nyle, at a cost of 10x increase in load.

6. **Asynchronous consensus without common coins.** We design QSC, the first practical asynchronous consensus protocol that does not rely on common coins. QSC assumes crash-silent faults. It builds on our novel group communication primitive, threshold synchronous broadcast (TSB), which provides step-synchronous message delivery atop an asynchronous network, and identifies a threshold of messages delivered at a threshold of nodes. TSB helps QSC be responsive and reduces design complexity via modularity. Thanks to TSB and to private randomness piggybacked on messages, we show that QSC nodes decide within $O(1)$ running time in expectation, which is optimal, and $O(n^3)$ total communication complexity in bits, but only $O(n^2)$ total message complexity.

Thesis scope. The scope of this thesis is as follows. First, the thesis addresses Lamport exposure at the application layer, namely the data and metadata dependencies of a key-value store and a blockchain architecture. We recognize that there may be complex cross-layer dependencies, such as higher up in the stack for web applications and lower down at the networking layer. We defer cross-layer dependencies to future work. Second, the thesis explores asynchronous consensus in the crash-fault model without common coins. We leave adapting QSC to the Byzantine model to future work.

Thesis statement. The following thesis statement encompasses the contributions:

This thesis argues that it is possible to systematically immunize systems to remote failures and partitions by limiting their Lamport exposure. Our key insight is to preserve the locality of interactions, not exposing them to distant failures, partitions or risks, no matter how severe. We substantiate our claim through our architecture of an exposure-limiting key-value store and a distributed ledger, and through simulations and experiments. We also show that consensus protocols can handle network asynchrony without relying on common coins, having the potential to make consensus more responsive and more practical. Our findings enable us to build systems resilient to failures, partitions, and slowdowns.

1.3.2 Thesis organization

This thesis is organized as follows. Chapter 2 presents the design and evaluation of Limix, our distributed metadata coordination service, that limits exposure to remote failures and partitions. The chapter also presents Lamport exposure metrics, the autozoning algorithm and its load scaling implications, how we maintain strong consistency. Next, Chapter 3 presents Nyle, our exposure-limiting trust-but-verify distributed ledger architecture. The chapter shows how Nyle's design empowers users to choose the zones they trust, but later verify their transaction and detect compromised zones. We also present an analysis of Nyle's security and a comparative simulation. Then, Chapter 4 presents QSC, our asynchronous consensus algorithm based on private randomness and TSB. The section includes the design of QSC, of TSB and safety and liveness proofs. Finally, in Chapter 5 we provide concluding remarks and future work directions.

2 A Globally-Managed Coordination Service Limiting Lamport Exposure

> All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection.
>
> David Wheeler

Globalized computing infrastructures offer the convenience and elasticity of globally managed objects and services, but lack the resilience to distant failures that localized infrastructures like private clouds provide. This chapter starts by explaining that configuration services face a fundamental challenge in providing *both* global management and resilience to distant failures. We then present the design of Limix, the first metadata configuration service to address this problem. Limix insulates global strongly-consistent data-plane services and objects from remote gray failures, network partitions, and slowdowns by ensuring that the definitive, strongly-consistent metadata for every object is always confined to the same region as the object itself. We continue with Limix's architecture and a detailed protocol for object lookup, which addresses consistency and migratory objects. Next we describe an autozoning-based control plane for Limix that provides availability bounds: any user can continue to access any strongly consistent object that matters to the user located at distance Δ away, while being insulated from failures outside a small multiple of Δ. We built a Limix metadata service based on *based on the key-value interface of* CockroachDB. Our experiments on Internet-like networks and on AWS, using realistic trace-driven workloads, demonstrate that Limix enables global management and commonly improves availability by at least 30% compared to the state-of-the-art.

2.1 Introduction

Organizations today face a choice between *localized* and *globalized* computing infrastructure, each alternative carrying important tradeoffs. Localized infrastructure hosted at the organi-

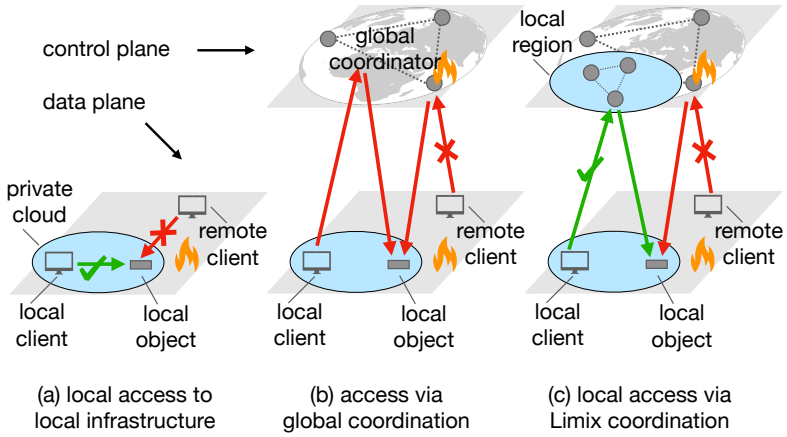


Figure 2.1: Lamport exposure in traditional (a) local or (b) global infrastructure, and (c) global infrastructure with Limix.

zation's own site(s), such as private clouds, carry higher internal management burdens but offer greater local autonomy, resilience to distant failures beyond the organization's control, and can be necessary to satisfy data privacy or digital sovereignty concerns. Globalized infrastructure such as public clouds, in contrast, offer many *global management* benefits: e.g., the convenience of instantiating objects or services on demand without worrying about their location, maximum elasticity in provisioning and adapting to changes in load, and the ability to migrate existing data and services without having to interrupt access or change their names.

Is it possible to achieve the local autonomy, failure resilience, and digital sovereignty benefits of localized infrastructure, together with the global management benefits of today's public clouds? Achieving such a "best of both worlds" appears fundamentally hard, in part because this choice boils down to a basic indirection conundrum. For clients to find and access localized infrastructure such as private cloud services, we can simply embed a locally-scoped or already "resolved" location directly into the identifiers of objects and services being accessed. Accesses by local clients to local objects can be simple, robust, and insulated from remote failures or network partitions outside the relevant domain, as illustrated in Fig. 2.1(a). But these objects are then "fixed" and cannot be migrated or managed globally without changing their names.

Global management, in contrast, requires "another level of indirection": typically a distributed coordination or metadata service, allowing clients located anywhere to discover the location and status of any object or service of interest. These services, however, expose clients to *gray global failures* such as network partitions [11, 61, 65], misconfigurations [2, 68], or cascading failures [11, 22, 65, 81], far beyond of the organization's geographic locality or domain of control – i.e., exposure to "the failure of a computer you didn't even know existed." This global failure exposure usually applies even when both the client and the target data or service are localized to the same network or region and have connectivity in the underlying

network [11, 22]. Even if the target data might be weakly consistent [44, 60, 80], metadata is usually strongly consistent for many reasons [13, 77] such as correct liveness determination, access control, and accounting. Dependence on globally geo-replicated metadata, however, can prevent even local clients from accessing local data if a majority of metadata replicas fail or become unreachable, as illustrated in Fig. 2.1(b).

To address this conundrum we introduce Limix, the first distributed coordination architecture that enables global management while also guaranteeing that localized accesses within a region of interest are insulated from global failures beyond that region, as illustrated in Fig. 2.1(c). Limix ensures that the definitive, strongly-consistent metadata for any data-plane object or service is always collocated in the same region as the object itself. Metadata in Limix thus enjoys a *fate-sharing* relationship [39] with both the target object *and* with any local clients accessing the object from within the same region – such as within an organization's own internal network, or within a relevant geopolitical domain such as a country. Metadata remains strongly-consistent and geo-replicated, but Limix confines the *definitive* replicas of an object's strongly-consistent metadata to the same region as the object itself.

Cell-based configuration services like Physalia [22] improve failure resilience for users within the same cell. These provider-managed cells, however, do not offer users direct transparency into or control over each user's *Lamport exposure* – the set of infrastructure components whose failure could affect the user – as discussed later in Section 2.2.1. Further, Physalia offers no guarantees for user activity that crosses cell boundaries.

Challenges and Contributions. Limix's design decisions for practical global manageability and resilience become apparent when we target applications where the locations from which data-plane items are accessed change dynamically. Data stores and locality management services with dynamic data access locality already migrate strongly-consistent data close to users [13, 77]. Limix ensures that users can continue accessing such nearby items under remote failures even during migration, while preserving strongly-consistent access.

Constraining the placement of strongly-consistent metadata in localities creates the further efficiency and scalability challenge of enabling any clients outside an object's current region to find the object without incurring the costs of either replicating all location information proactively across all regions, or potentially having to search all regions during any metadata query. Limix builds on techniques from compact graph summarization theory [115, 116] to limit the bandwidth and processing costs of these global searches to a small multiple of the baseline cost of querying a single global metadata service. The metadata-access costs of Limix's failure insulation is thus only about 2× in the common case of an object administratively localized to a single region. Since metadata query costs usually represent only a small fraction of the total "end-to-end" costs of accessing most data-plane services, a 2× metadata query cost increase is insignificant overall to most applications, and is much lower than the $N\times$ metadata cost increase that cell-based architectures with N distributed discovery services (or the proactive replication of location hints across all N regions) would otherwise incur.

Further, Limix ensures by design that not only availability but also metadata access latencies observed by local clients are insulated from global outages or slowdowns, and that they closely reflect the best local communication latencies available on the underlying network.

To evaluate Limix's applicability and performance, we prototyped a Limix configuration service for an exposure-limiting key/value store. For metadata/configuration storage, our prototypes use CockroachDB [77], a widely-used, strongly-consistent distributed data store. Our experiments running realistic workloads based on metropolitan traffic traces on AWS, and on a testbed simulating realistic scenarios, show that Limix outperforms Physalia's availability during reconfigurations while providing strong availability guarantees. Our experiments further explore the tradeoffs between Limix's overheads and availability guarantees: at scale, the dynamic load overhead is logarithmic in the number of nodes and network width.

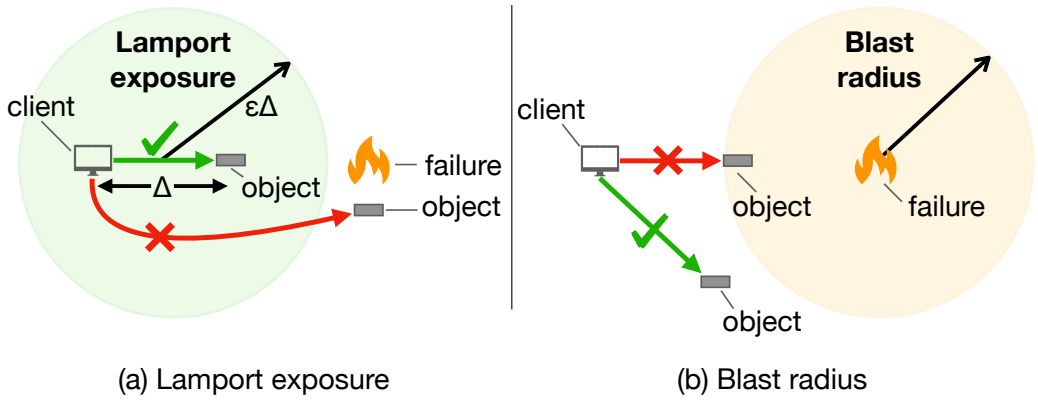
In summary, the contributions of this chapter are as follows. Contributions two, three and four may be of independent interest for other systems that aim to define and limit exposure.

1. The design of Limix, the first distributed metadata coordination service that enables global management while protecting local accesses from distant failures.
2. A definition of Lamport exposure, along with several metrics to quantify it.
3. The construction of exposure-limiting zones, which provide the structure and granularity for systematic exposure control.
4. An autozoning scheme ensuring by design that a user accessing any data at an exposure-distance Δ away is protected from all failures occurring beyond a small multiple of Δ.
5. A prototype implementation of Limix and Physalia on top of CockroachDB with a comparative evaluation.

This chapter is based on joint work with Georgia Fragkouli, Enis Ceyhun Alp, Michael F. Nowlan, Jose M. Faleiro, Gaylor Bosson, Kelong Cong, Pierluca Borsò-Tan, Vero Estrada-Galiñanes, Bryan Ford, published in Băsescu et al. [23, 24].

2.2 Context

This section gives the necessary background for a strongly-consistent configuration service like Limix. We first explain that the CAP theorem imposes restrictions on the available Limix can achieve, and that Limix does not conflict with the CAP theorem when prioritizing availability of local accesses. Second, we focus on Limix choice to prioritize user-perceived availability, for the data-plane objects that matter to users. Reducing the Lamport exposure, which is the user-centric viewpoint of Limix, contrasts to Physalia's provider-centric viewpoint, i.e., blast radius. Our discussions with risk-sensitive customers share Limix's viewpoint. Finally, we argue that access locality is prevalent in globalized applications, and thus, Limix's focus on shielding local user activity leads to a sizeable increase in user-perceived availability.



(a) Lamport exposure (b) Blast radius

Figure 2.2: Illustration of Lamport exposure and blast radius.

2.2.1 Lamport exposure and blast radius

Taking inspiration from Leslie Lamport's quote in Chapter 1, we informally define the *Lamport exposure* of any given user U as the set of computing infrastructure – i.e., every "computer U didn't even know existed" – that "can render U's own computer unusable". Lamport exposure is thus meaningful only with respect to the activities of some user U.

Limix seeks to place a strong bound or "shield" on the Lamport exposure of any user U whenever U accesses data or services that are *local*, i.e., close to U by any suitable distance metric. We may define locality based on administrative boundaries such as those of an organization or a country, or via metrics such as round-trip time (RTT). The availability of U's local accesses should be unaffected by remote failures and partitions beyond Limix's Lamport exposure shield: not just by individual server failures but also by network partitions and *gray failures*, or partial failures that can cascade into correlated failures and partial partitions. Gray failures have a multitude of causes including malfunctioning switches and software bugs that partially drop traffic or prevent simplex communication [11, 61, 65, 81]. Figure 2.2(a) illustrates a Lamport exposure bound shielding a user's accesses to an object at distance Δ from failures beyond a distance $\epsilon\Delta$, for some small factor $\epsilon \geq 1$ (ideally 1, but this may be unachievable).

Locality matters. By limiting Lamport exposure, Limix seeks to offer availability guarantees to users for the data and services that matter most to them, which are often local. It is already common for data stores to migrate data close to users to improve performance [13, 77], even without guaranteeing availability. Privacy and sovereignty considerations often motivate organizations or governments to require a user's "data at rest" to remain within the user's country, or a region such as the EU – even if these policies cannot currently ensure that this data will *remain accessible* despite outages beyond the relevant borders. Similar considerations motivate many organizations to confine their most-critical data and applications to local infrastructure such as private clouds, giving up the benefits of global management. In general, people more willingly trust organizations and services perceived to be more local [4, 6]. We hope that Limix might enable providers to offer services to more locality- and sovereignty-

conscious users who might currently avoid globally-managed infrastructure entirely.

In contrast to Limix, Physalia [22] is a cell-based architecture that aims for higher availability by limiting the *blast radius* of a failure. Blast radius represents the system components and objects that could be affected by the propagation of a failure or partition (Fig. 2.2(b)). Blast radius is thus a complementary concept focused (or "centered") on the location of a *failure*, rather than on the location of a *user* potentially affected by it. From an infrastructure provider's perspective, reducing blast radius can reduce the number of users affected by any single failure. Being focused on the locations of failures and heavily dependent on internal details of the provider's infrastructure, however, blast radius does not offer obvious guarantees that appear directly meaningful to users.

2.2.2 Coordination and the CAP theorem

Strongly-consistent coordination systems (Zookeeper [67], etcd [3]) are an essential building block for large scale distributed applications. Distributed applications replicate their state in order to enhance resiliency to failures, and to decrease latency through proximity to clients. But an unsought side-effect is the need to coordinate these replicas. Hence the need for coordination services: These provide a basic set of operations – such as liveness determination, correct identification of the replicas storing a data item, lease holders, access control, accounting, etc. Because these functions need to be *correct*, coordination services implement consensus among the configuration replicas, ensuring strong consistency of the configuration.

Coordination systems often are not critical to application availability until failures occur. Applications routinely bypass the coordination system for configuration reads using leases [25, 77]. During partitions or failures, however, the data plane cannot bypass the configuration system, because it needs to reconfigure its data replicas. This is when the configuration system becomes critical to application availability and performance. Reconfiguration requires strongly-consistent configuration writes to agree on the new configuration. Until reconfiguration completes, the data-plane may be partially (e.g., operate in read-only mode only) or fully unavailable.

The requirements of configuration systems to be strongly consistent and available seem to conflict with the CAP theorem [58]: under partitions, a strongly-consistent (configuration) system cannot remain available (on both sides of a partition). However, Limix does not conflict with the CAP theorem when prioritizing availability of local accesses. Remote users may not be able to access remote data during partitions, but local users can, without breaking strong consistency, and in many cases as described above, local data is what interests users. For this reason, Limix collocates metadata with its data.

To decrease the risk of being affected by remote gray failures, Brooker et al. [22] suggest many smaller configuration service deployments instead of a network-wide "monolith". Deploying several configuration services instead of a single globalized deployment is one of the princi-

ples that Limix also applies. However, as opposed to deploying disjoint cells, Limix creates overlapping, redundant configuration service deployments, organized to provide availability guarantees for any user accessing any object or service.

2.2.3 Perceptions of risk-sensitive customers

To obtain early feedback on the motivation for this work, we held informal discussions with two representatives from a large non-profit organization and a global company, respectively, who are familiar with their organizations' current use of – and appetite for the use of – cloud computing technologies. Our goal was to learn from them how the availability of cloud services affects their organizations and perceived needs, particularly across jurisdictions. An important goal of the discussions was to expose the representatives to the basic of Limix's design, but only later on in the discussion, after learning their unbiased opinion. Though not comprising a formal study of any kind, the feedback we received uniformly suggested that service sovereignty is a desirable feature for their critical operations — one representative even called it the "holy grail".

Both representatives confirmed that their companies are increasingly migrating applications to globalized infrastructures such as public clouds, to take advantage of elasticity and lower management effort. Each discussed applications that span multiple jurisdictions, but with the requirement to manage data and data services differently in different states, countries, or regions. They also confirmed that in some cases where data security is critical, data sovereignty and regulatory requirements prevent them from using cloud services at all.

We discovered that for cloud-based services, a service-level agreement (SLA) is usually insufficient to align client expectations with their service providers. Some organizations that are not "big enough" believe they have little influence over SLA specification. Whereas some companies that may afford to pay more for a custom tailored SLA that covers their critical services are concerned about whether and how these SLAs are enforceable. Furthermore, they are concerned about becoming "locked in" to a specific cloud provider. The fear of being "locked in", different legal requirements across jurisdictions, privacy and reputation concerns, and the perception that outages are still uncomfortably high are some of the reasons why the representatives were reluctant to trust and fully embrace cloud infrastructures.

Companies want cloud portability. One representative would prefer to treat clouds as bare-metal infrastructure, and if a service fails on one cloud, it should be possible to move it to another. The representatives proposed that by replicating data, services, and applications, service sovereignty may enable portability.

Both representatives see the need for a localized implementation growing in tandem with the trend toward a more distributed workforce. They see this not only for large global corporations, but also for more local small and medium-sized businesses, for whom a service that limits Lamport exposure makes a lot of sense given the relative locality of their typical business, for

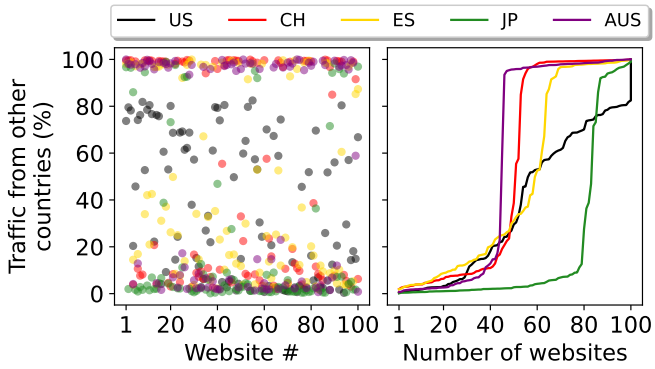


Figure 2.3: Prevalence of access locality. Top 100 websites in 5 countries and their traffic percentage from other countries (individual websites and CDF).

which they do not want global exposure. Both the representatives would consider leveraging a service that limits Lamport exposure for their critical services: they indicated they would be interested in the service, despite expecting a higher price than today's services. The reason is that, for them, Lamport exposure guarantees are easier to understand and more meaningful compared to limiting the "blast radius" of a failure.

In conclusion, while these informal discussions do not purport to be a systematic study, they do appear to indicate that locality — particularly the ability to guarantee maximum availability within a relevant locality — does matter to many customers and potential customers of cloud computing services.

2.2.4 Estimating prevalence of access locality

We observe that many applications and services exhibit bimodal access locality. A high proportion of accesses are by users mostly in a given country or other region representing the application's primary customer base or target audience. Other users of these same applications and services, however, tend to be globally distributed, accessing the service from anywhere (e.g., roaming employees or expats). Thus, applications must efficiently support global accessibility for global users, while prioritizing maximum availability and performance for local users representing the most critical target customers.

To test this bimodal-access hypothesis, we use the top 100 websites of five OECD countries (namely United States (US), Switzerland (CH), Spain (ES), Japan (JP) and Australia (AU)) as rough proxies for applications, and examine the access distributions they exhibit. We obtained the data from Similarweb [105], a company specializing in web traffic and performance. Since these are the most visited websites in their respective countries, we conjecture that they have a strong local presence. In Figure 2.3, we show that these websites also have a global presence, as 17–56% (JP–AU) of the websites receive at least 50% of their traffic from outside of that country. The results of our simple study does not make any assumptions about the consistency models used by the websites, but it suggests that locally-prevalent applications are indeed globally

relevant as well.

2.3 Setting and Goals

In this section we discuss the main system components, our assumptions and our goals.

Limix involves the following concepts:

Items. Limix is a configuration service for existing data-planes, such as a key-value store as in our prototype (Section 2.7). We refer to the data as *items*, defined at the granularity at which the data-plane client may access the data, e.g., key-value pairs. Limix facilitates the lookup of items by managing the configuration information that stores each item's location. Limix keeps the metadata up-to-date by interfacing with the existing data store to react to item creation/migration/reconfiguration by creating/migrating/altering the configuration.

Sites. Sites are the nodes that deploy Limix, which we assume to be connected through a network. For example, sites may be data centers, which is the setup we explore in our prototype and evaluation. Limix can seamlessly run on sites managed by multiple service providers.

Clients requests. Clients interact with Limix by making item lookup queries to a site. Once the client request reaches a site, Limix limits the query's exposure. Limix does not improve last mile availability, such as when a client cannot reach any site. We make no assumption on the site a client chooses. Clients can send lookup queries to any site they wish, e.g., when the client's location changes or for other reasons, and the lookup results are always correct. Limix gives the client exposure-limiting guarantees w.r.t. the location of the client-selected site. Therefore, it would be prudent for clients to choose nearby sites for their queries as measured by the Lamport exposure metric.

Zones, authoritative zone, definitive replicas. Limix provides exposure-limiting guarantees at the granularity of *zones*. A zone covers the set of sites that a client may depend on when looking up items in that zone. Because the data plane is zone-agnostic, Limix tracks the item location w.r.t. zoning and always collocates the item configuration with the item. The location of an item, which we call the item's *authoritative zone*, is the smallest zone holding a quorum of the item's data plane replicas. Then, inside the same zone, Limix stores the most the most recent configuration (metadata) for the item, called the *definitive metadata replicas*. In contrast, non-authoritative zones may have only eventually-consistent metadata replicas for that item.

Goals. Limix has the following objectives:

- *Availability guarantees.* Provide strong availability guarantees that might be legally or contractually mandated to hold at all times, even during item migration.
- *Simultaneous constraints.* Satisfy simultaneous sovereignty and locality constraints. For example, a Germany constraint is more restrictive in placement, while an EU constraint

protects a larger set of users.

- *Load balancing.* Spread workload for item lookup e.g., avoid overloading small zones with global accesses. A user imposes load only on the zones the user's site is in.
- *Dynamic data plane.* Enable dynamic data planes to migrate data routinely, without restricting them with e.g., static partitions of data across regions.
- *Strong consistency.* Enforce that item lookup returns strongly-consistent items, or be unavailable for that item if the latest item version is not reachable.
- *Autozoning.* Provide an automatic zoning capability, which enforces locality constraints for all users, the vast majority of whom just "want things to work." We desire reasonable but fully-automatic risk-limiting policies requiring no specific understanding of the workload or administrative effort.

2.4 Design

This section outlines Limix's design in a step-by-step fashion for clarity. We first list Limix's challenges, then introduce Limix's per-zone configuration service, and explain how it limits its maintenance loads on local zones. We then address the problem of satisfying multiple simultaneous exposure-limiting constraints on one item, ensuring that lookup replies follow data-plane consistency, and handling item migration.

2.4.1 Challenges

The goal of local availability under global management, coupled with the requirement strong consistency for metadata, imposes key challenges on Limix's design.

Consistency. Because data and services must still be movable and accessible from anywhere under normal conditions, Limix must enable clients anywhere to locate a data item's current definitive state globally, while insulating local clients from global failures. This requirement implies replicating location information simultaneously across global and local deployment zones, which in turn exacerbates consistency challenges (Sections 2.4.2 and 2.4.3).

Load balancing. We must ensure that control-plane queries about globally-popular data do not overload a small, lightly-provisioned zone it may be located in. Limix addresses this challenge by systematically ensuring that each zone, local or global, serves only clients querying the service from *within the same zone*, and can therefore spread the access workload without risking overload from external queries (Section 2.4.4).

Simultaneous item constraints. Data plane items may have to satisfy more than one locality or sovereignty constraint – such as that local clients in Germany be insulated from failures outside Germany, *and* that all clients in the EU be insulated from failures outside the EU. This goal requires that Limix allows state replication across multiple *overlapping* zones (Section 2.4.5).

Item migration. Limix must maintain both strong consistency and local availability even

during object migration: ensuring, for example, that data migrating from Germany to France remains immune to failures outside the EU even during the transition. To address this challenge, Limix uses a multi-phase process to migrate the data's definitive state while maintaining eventually-consistent location hints in larger zones beyond the data's origin and destination (Section 2.5.3).

2.4.2 Item lookup overview

For zones to act collectively as a unified system, Limix needs to enable clients to find data that can be located anywhere, regardless of the client's zone. However, robust and efficient item lookup is challenging. Fig. 2.4a illustrates a straightforward but inadequate approach, relying on a central service to store the configuration for item lookup. This service increases the client's exposure beyond the perimeter of the client's and data's common zone. The single zone may also become overloaded with requests from all zones.

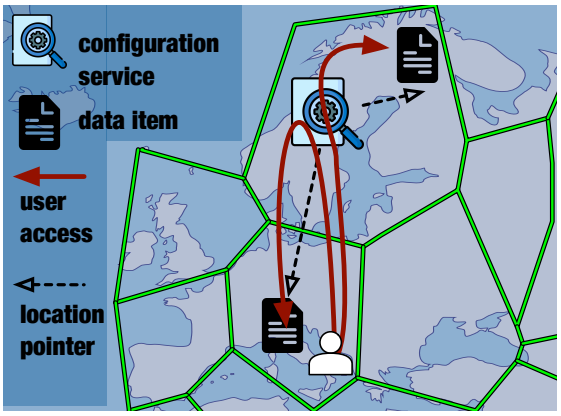
Of course, this strawman could be easily made scalable by distributing the configuration service across many/all zones, using standard techniques such as consistent hashing of keys. However, consistent hashing still increases a node's Lamport exposure beyond the zone boundaries. Consider a client requesting data without having the data location cached. To resolve the location, the client might need to query configuration service nodes in zones different from the zone holding the requested data. Partitions might prevent the client from reading the data location, even though they might not isolate the client from reaching the data itself. The problem with this approach is that data and corresponding metadata are not collocated in the same zone, and hence lack fate sharing [39].

Limix thus needs to ensure that a client in a given zone can always find an item within the same zone using *only* resources within that zone. Efficiently collocating data and metadata in the same zone so that they have the same Lamport exposure represents the first challenge for Limix, which we address by having a distributed configuration service *per zone*.

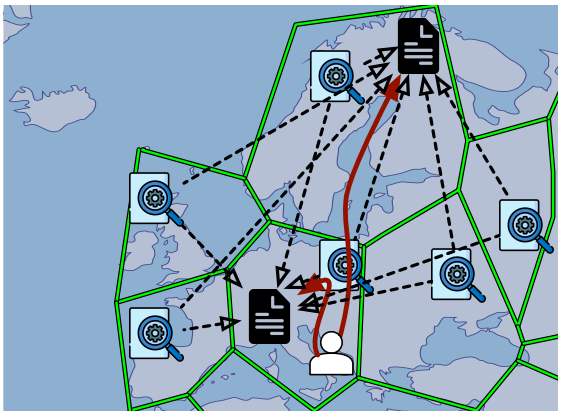
2.4.3 Configuration service consistency

As a next strawman approach addressing the challenge above, we could replicate *all* lookup pointers in *all* zones, as depicted in Fig. 2.4b. However, this strawman introduces a consistency challenge: Because all zones' configuration services store pointers to all items, when an item is deleted or migrated, regardless of the zone where the item resided, all configuration services in all zones should be updated with the new pointer. If we required strongly consistent state for all zones' configuration services, we would increase an item's exposure to all zones, which is undesirable.

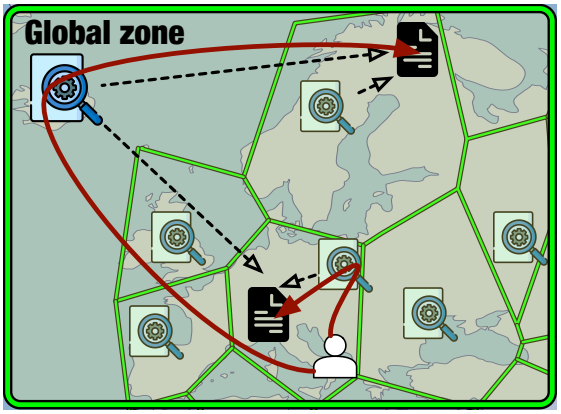
Limix addresses this exposure challenge as follows. Each zone's configuration service stores strongly consistent pointers only for the items inside the zone. If an item's configuration changes, for example, only the configuration service in the item's zone needs to update the



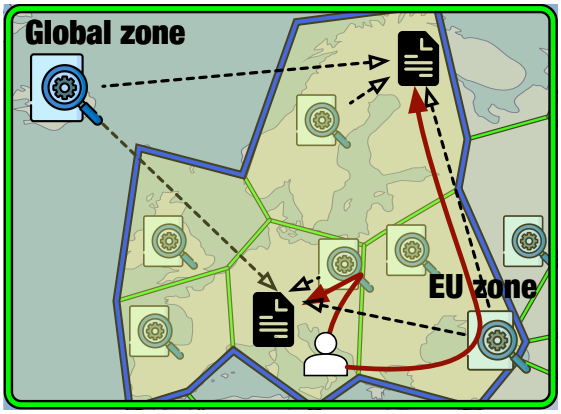
(a) Global configuration service, holds pointers to all items.



(b) Full configuration service per zone, each holding pointers to all items.



(c) Local configuration service per zone, each holding pointers *only* to local items.



(d) Limix configuration service with overlapping zones, each holding pointers to local items.

Figure 2.4: Strawmen for the configuration service and their management of pointers to data items.

item's definitive configuration (i.e., location metadata) (see Section 2.3) immediately, on the critical path. Other zones' configuration services may update their metadata lazily with eventual consistency, outside the critical path of the item reconfiguration.

With this approach, each zone has its own configuration service that stores "location hints" for where an item was last known to be located. But this strawman invites a second challenge: How do clients locate items given that configuration services might (temporarily) store outdated pointers? We distinguish two causes of outdated pointers. The first reason is item migration, when Limix updates pointers outside the critical path. Could a client be unable to find an otherwise reachable item when the item migrates? Limix addresses the challenge by temporarily storing a *Permanently moved to* marker at the item's old location. On encountering this marker, a client follows it to the new location. Limix prevents long indirection paths by eventually updating all pointers, after which it deletes the marker. The client stops following pointers when it reaches the item's authoritative zone (see Section 2.3); Limix coordination ensures there is a single authoritative zone per item. Section 2.5.3 provides a detailed description.

The second reason for items to be outdated is during partitions. If Limix cannot reach some zones' configuration services, pointers will be stale. The main challenge is to ensure that clients do not return stale items because of stale pointers, which would break strong consistency. Limix clients rely on authoritative zone indicators, as explained above, to decide whether the pointers point to the most recent item version. However, there is one remaining issue when partitions heal and several migrations are in place: Pointer updates might arrive out-of-order, causing an old update to overwrite a newer one. We use versioning for pointers to avoid this situation. Every pointer update increases the pointer version and a pointer update occurs only if the update has a higher version than the existing pointer. The update makes use of the compare-and-swap primitive offered in the API of most strongly-consistent KV stores. Section 2.5.3 provides a detailed description.

2.4.4 Lookup load on (small, local) zones

The above strawman invites the question: What is the load on each zone's configuration service? Consider updating the configuration service after an item insertion or migration. Either the destination zone could push the new item location to all zones, or the client's zone could pull the item location on demand. Both approaches incur $O(n)$ load and communication overhead *per client request* for n zones.

Limix instead spreads the lookup loads and limits query burden on small zones by organizing a default overlapping global zone. In our next strawman illustrated in Fig. 2.4c, local zones store the location only for their local items and serve lookup requests only from local sites. In contrast, the global zone serves as the backup reference point, whose globally-distributed configuration service knows any item's location. Every zone propagates location updates to the global zone.

A client queries only its own local configuration service and the global one, without overloading other small zones. Thus, instead of $O(n)$, overhead per update becomes $O(1)$. The global configuration service must service load from all $O(n)$ zones, but it has server capacity distributed across all $O(n)$ zones among which to share that load. Location updates propagate only eventually, off the critical path, to limit the source zone's exposure to failures beyond its borders.

2.4.5 Item placement and zone overlap

Aside from the global zone, we assumed so far that local zones are disjoint. This assumption has a significant limitation, however: it cannot support *simultaneous* exposure-limiting policies, which may apply by law or contractual obligation. An item located in Germany may need to be accessible by clients in Germany with Lamport exposure limited to sites within Germany, *and* ensure that any client in the EU can access the same item with exposure limited to the EU. With the above strawman configuration service, unfortunately, EU clients outside Germany

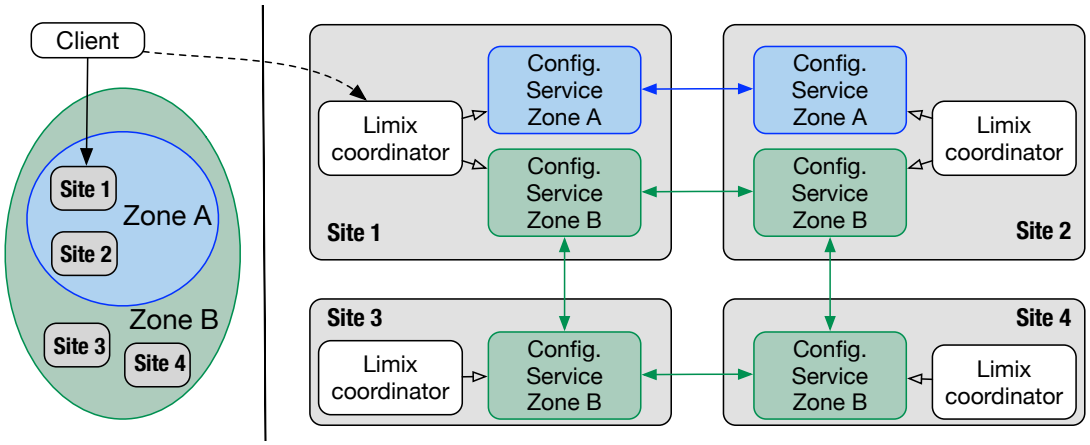


Figure 2.5: Limix's architecture on a simplified example with four sites and two zones.

must query the global configuration service, yielding global (not EU) Lamport exposure.

We address this problem by allowing local zones to overlap. All zones have a configuration service, and every zone propagates location updates to larger overlapping zones, up to the global zone. The global zone still acts as a master reference, holding pointers to all items. Update overhead increases slightly compared to the single global zone, from $O(1)$ to $O(v)$, where v is the maximum overlap depth. However, this approach limits Lamport exposure to smallest zone containing both the item and the client accessing it. Location updates propagate only eventually, outside the critical path, to limit the source zone's exposure to failures outside its borders. Fig. 2.4d illustrates the final configuration service architecture.

2.5 Limix Architecture and Lookup Protocol

This section presents Limix's architecture and protocol for item lookup and metadata management. We first present the architecture, showing the Limix components that run on each site, how these components interact with each other, and how a Limix client issues lookup queries. Then we present item lookup in a simplified scenario where items do not change location. Afterwards, we enhance the configuration service with consistency guarantees, such that item lookup locates the latest version of an item even during migration.

2.5.1 Architecture

Fig. 2.5 depicts the Limix architecture on a simplified example with four sites and two zones. Each Limix site runs a *Limix coordinator* (Algorithm 1), which interposes between the client requests and the Limix configuration services (CS). The coordinator exposes the itemLookup API to the client. Through upcalls from the data store, the coordinator associates the item location with zones.

Limix runs a CS per zone, thus all sites part of the zone collectively run that zone's CS. For

Algorithm 1: Limix main structures.

```
// Every site has a Limix coordinator, exposing the following API
struct SiteCoord
    ZonePtr, PointerVersion itemLookup (K)          // Locates the authoritative zone and its
      configuration version for item K.
    UpdatePointersOnMigration (K, SrcZone, DestZone)    // Upcall from data store to update
      configuration when item K migrates from SrcZone to DestZone.
    ConfigSvcs GetAllConfigSvc()
// A Limix zone stores the configuration and offers the typical API of a strongly consistent
   KVStore for pointer reads and writes
struct ConfigService extends KVStore
    ZonePtr, PointerVersionRead(K)                  // Reads the configuration for item K.
    WritePointerCAS (K,NewConfigValue,Comparator)   // Atomically swaps using CAS (compare and
      swap) key K's prior value with NewConfigValue if Comparator evaluates True on the prior
      value
```

example, SiteCoord_1 is part of two zones Zone_A and Zone_B, and correspondingly runs two configuration services: CS_A and CS_B for Zone_B. Whereas SiteCoord_3, part of Zone_B, runs just CS_B. Each CS is a KVStore that stores metadata about items (Algorithm 1).

The modular structure and interactions between components are fundamental for limiting exposure. First, there is no cross-configuration service communication. A CS process of a zone running on a site interacts *solely* with other CS processes for the same zone (running on other sites). This implies that, when a zone's CS answers a query, it relies only on zone-internal sites. Thus, there is no exposure to sites outside the zone. Second, the coordinator of a site issues on behalf of the client independent lookup requests to different CS. If a request fails because a zone is partitioned, other requests are not affected if they run on unpartitioned zones. This design decision also supports load balancing: the coordinator, only issues requests to CS running on the site, not other CS services. This means, the coordinator of a site does not put load on zones that the site is not part of.

2.5.2 Item lookup protocol

We now describe item lookup in a simplified scenario where items are immutable and do not change location, and the pointers of the CS in each zone are up-to-date. In this scenario, because items are immutable, we do not need to worry about pointer or item consistency. The next section (Section 2.5.3) addresses pointer update when items migrate, and pointer consistency when items are updated.

Algorithm 2 describes item lookup. A client calls the lookup function itemLookup on a site's coordinator, passing the item key k as a parameter. The coordinator sends parallel lookup queries to all CS running on the site, in search for the authoritative zone of k. Each CS reads the pointer for item k, which indicates a zone that the CS knows to have the item. In general, these pointers might be outdated due to item migration, and the search can follow multiple pointer indirections. However, by our assumptions for now, all pointers are up-to-date, thus

Algorithm 2: itemLookup($\text{SiteCoord}_{\text{client}}, k$) implements the lookup of item k submitted by the client at $\text{SiteCoord}_{\text{client}}$.

Input : $\text{SiteCoord}_{\text{client}}$, the site contacted by the client
k, the key of the item the client looks up
Output: AuthZone, the authoritative zone of item k
ItemVersion, the latest version of the item at lookup time

```
1  AuthZone, ItemVersion ← ⊥, −1
2  for CS_i ∈ SiteCoord_client.GetAllConfigSvc() do   // Iterate (in parallel or serially for lower load)
   through all CS_i running on SiteCoord_client
3      while CS_i.IsAuthoritative(k) = False do          // Stop when we find the authoritative zone
4          PointerToZone, _ ← CS_i.Read(k)               // Recursively follow location pointers
5          if PointerToZone ≠ nil then
6              CS_i ← PointerToZone.GetConfigSvc() // Obtain the CS of the zone the pointer indicates
7          else
8              break                                 // Stop when CS does not have the location of item k
9      if CS_i.IsAuthoritative(k) = True then
10         AuthZone, ItemVersion ← CS_i.Read(k)   // Update AuthZone and ItemVersion if we found the
           authoritative zone
11 return AuthZone,ItemVersion  // AuthZone can be nil when the authoritative zone is not available
```

when a CS indicates a non-nil zone for item k, it will be the AuthZone for the item

A CS could have a nil pointer for item k, even when all pointers are up-to-date. This is because, by design, not every zone's CS needs to know about all items. However, the guarantee is that, as long as the network does not partition the item away from the client, there is at least one CS that the coordinator queries and that knows the item's location.

Why does item search not degrade the exposure guarantees? Each of the parallel lookup queries accesses the zone-private CS, having dependencies only inside the zone. Other parallel searches might hang in case of partitions or slow performance. However, because each search executes and completes independently, parallel searches do not affect each other. Of these zones, the ones that reply first with a pointer chain leading to the authoritative zone determine the client's exposure for that item. If at least one such set of zones is partition-free, the client is *guaranteed* to find the item. Thus, the coordinator bounds a client's exposure to the smallest zone of the client that contains the item.

Optimization. It appears possible to reduce load by "pacing out" lookup requests to zones, a likely desirable optimization in practice. A client can send the same lookup request to one-zone-at-a-time, starting with the closest zone, only contacting other zones if the client fails to receive a reply within a given timeout. Limix still needs to run all configuration service to be able to continue functioning when failures and partitions do happen. But this technique may significantly reduce the runtime load on these zones. Pacing out requests likely incurs minimal overhead not only in "normal-conditions" when failures and partitions do not disrupt zones, but also for data predominantly accessed in a few close localities. This optimization makes a tradeoff between overheads and how fast the replies are, but maintains the same availability guarantees.

Algorithm 3: UpdatePointersOnMigration is an upcall from the data store to the Limix coordinator. It implements the pointer update for item k when it migrates from SrcZone to DestZone.

Input : k, the key of the item that migrates
SrcZone, the zone the item migrates from, which is by definition the authoritative zone
DestZone, the zone the item migrates to

```
1  SrcCS ← SrcZone.GetConfigSvc()                                  // The CS of the authoritative zone SrcZone
2  _, OldPointerVersion ← SrcCS.Read(k)        // SrcCS has the most recent pointer version for item k
3  PointerVersion ← OldPointerVersion + 1
4  SrcCS.CASWrite(k, DestZone, PointerVersion)     // Pointer marker for item k in SrcCS pointing to
     DestZone; avoids out of order writes
5  DestCS ← DestZone.GetConfigSvc()                                                  // The CS of DestZone
6  DestCS.CASWrite(k, True, PointerVersion)          // By updating the pointer for item k in DestCS,
     DestZone becomes the authoritative zone DestZone; avoids out of order writes
7  in parallel do          // Do in background pointer updates for outer zones, which cuts latency by
     eliminating chains of pointer indirections, but is not needed for consistency
8  |  UpdatePointersOuterZones(k, SrcZone, False, PointerVersion, <)       // Update pointer for outer
   |    zones of SrcZone
9  |  UpdatePointersOuterZones(k, DestZone, DestZone, PointerVersion, ≤)       // Update pointer for
   |    outer zones of DestZone
10 SrcCS.CASWrite(k, nil, PointerVersion)      // Background garbage collection of the marker in SrcCS
     after all pointers were updated
```

Algorithm 4: CASWrite implements CAS-based write based on version numbers.

Input : CS, the coordination service that applies the write
k, the item for which the CS_u pdates the configuration
v, the new configuration value
ConfigVersion, the version of v
Comparator, a comparator function that compares two versions

```
1  NewConfigValue ← v||ConfigVersion            // NewConfigValue contains both v and ConfigVersion
2  CS.WritePointerCAS(k, NewConfigValue, Comparator(CrtConfigVersion, ConfigVersion))      // Update
     the configuration for item k with NewConfigValue only if Comparator returns True
```

2.5.3 Lookup during item migration

The placement of items may need to change, for example due to client-perceived performance and load-balancing algorithms. Or this could be caused by a policy change, e.g., a new constraint on which existing zone(s) a particular item is allowed to be placed in or migrated to. Because items involve strongly-consistent state, a challenge is that Limix must maintain strong consistency for configuration during item's migration. Three questions arise at this point: (a) How can we ensure that the coordinator locates the latest version of an item, even during migration or partitions? (b) Given that clients could update data plane items anywhere in the system, how do we ensure another distant client, located in a different zone, finds a specific item? (c) How do we ensure item search does not degrade the exposure-limiting guarantees?

Algorithm 3 depicts item migration from the item's authoritative zone SrcZone to DestZone. When Limix receives an upcall from the data store that an item is being migrated, it does the following: (0) Record the most recent pointer version, taken from the item's authoritative

Algorithm 5: UpdatePointersOuterZones implements CAS-based configuration update in all zones containing the reference zone z.

Input : z, reference zone k, the item for which the CS_u pdates the configuration
v, the new configuration value
ConfigVersion, the version of v
Comparator, a comparator function that compares two versions

```
1 for OuterZone_i ∈ z.getOuterZones do          // Iterate through outer zones of z
2     CS_i ← OuterZone_i.GetConfigSvc()          // The CS of OuterZone_i
3     CS_i.CASWrite(k, v, ConfigVersion, Comparator)   // Use CASWrite for the configuration update
```

zone, i.e., SrcZone, and increment it (lines 2-3). We use pointer versioning and compare-and-swap for the pointer updates below to avoid old pointers overwriting newer ones (CASWrite in Algorithm 4). (1) Update the pointer in the old zone with a forwarding reference, indicating the item is being migrated to the new zone and the old zone should no longer be considered authoritative (line 4); (2) After item migration, update the pointer in the new zone indicating migration is complete and the new zone is now authoritative, meaning that item is now usable at its new site (line 6); (3) In parallel and outside the critical item lookup path, update discovery service information in all zones for the item's old and new location (lines 7-9); and (4) Garbage collection: The item's configuration may finally be deleted in the old zone (line 10).

The algorithm above guarantees that any client can locate an item, even during/after migration. After step 1, all clients following existing pointers for that item (using Algorithm 2) that lead (directly or indirectly) to the former authoritative zone learn that the item is being migrated to the new zone. After step 2, when the item finishes migrating, all clients following existing pointers locate the new authoritative zone.

Up to now, clients can find the item, but through a potentially longer chain. For example, if the item moved from Germany to France, clients in the US might first find the pointer to Germany and then follow it to France. However, eventually, when all pointers in step 3 finish updating, all clients in the system can locate the item by following a single pointer. The exposure during migration is proportional to the RTT between the client and the two zone locations of the object. Specifically, if the item migrates $Z_{item} \rightarrow Z_{new}$ then a client's exposure for accessing the item *during migration* is $Z_{client} \cup Z_{item} \cup Z_{new}$, and $Z_{client} \cup Z_{new}$ after the migration. Importantly, because pointer updates happen independently and in parallel, partitions cannot disrupt a client to locate the item, as long as both the client and the item share a non-partitioned zone, which is guaranteed to exist within a small RTT from the client and the item (Section 2.6.3).

2.6 Control Plane Zoning

This section provides Limix's control plane that builds zones to insulates users accessing items from distant failures and partitions. We begin by precisely specifying what "distant" means. In that sense, we define several exposure metrics that may be appropriate, depending on

the situation. Some metrics provide by nature an explicit zoning, for example jurisdictions, e.g., Germany, the EU, the World, which are useful when items are constrained by regulatory bounds. However, for other metrics zoning may not be explicit. For these, we present an autozoning protocol that can use any continuous exposure metric, such as network latency in our design, and provides formal availability guarantees. We argue that, for any user accessing an item of interest from a distance Δ, autozoning limits the access's exposure – availability and performance – to a perimeter of at most $O(\log N) \times \Delta$ around the item.

2.6.1 Metrics for Lamport exposure

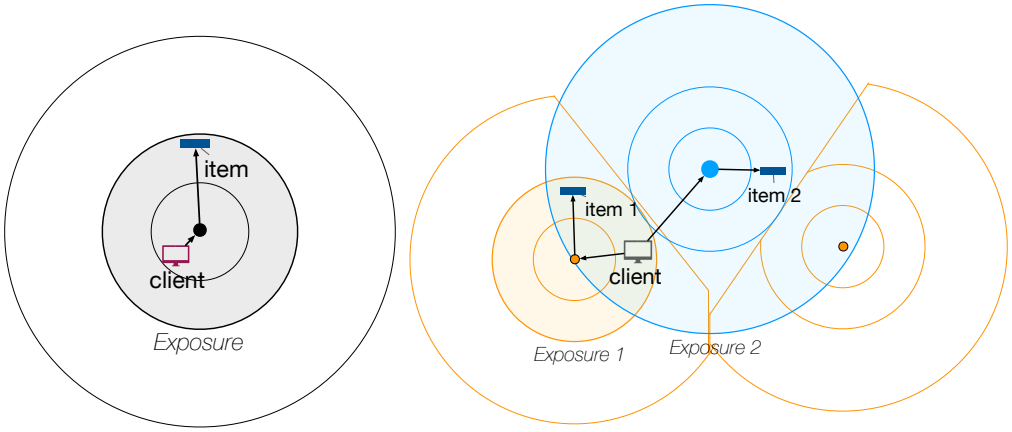
As the discussion in the previous sections suggests, one possible exposure leverages administrative or legal boundaries, such as countries or economic areas. This metrics provides explicit zoning, which enables complying with predetermined requirements, like regulations. For data or services in Germany, for example, regulatory considerations might demand a metric that treats all sites outside the EU as more distant than other sites in the EU, independent of geography.

Other metrics provide a continuum for measuring exposure, without any predefined zoning. An obvious metric is geographic straight-line distance. Because users within geographical proximity tend to interact more [102], this metric improves the resilience of common-case interactions. The first two metrics could use public geo-location databases to define zones, yielding static zones incurring low administrative overhead. A third distance metric is round-trip time (RTT), particularly relevant for time-sensitive use cases such as gaming and voice calls. A fourth metric might be network hop count, on the grounds that network paths with fewer hops are likely to have fewer hidden dependencies and thus be more resilient.

Limix using RTT as exposure metric. Limix uses round-trip time (RTT) as the exposure metric, and uses it to define zones. A zone's RTT diameter defines the zone's exposure: a lower RTT diameter means a lower exposure to remote failures, thus higher availability. During bootstrapping, we build an inter-site RTT map: the sites measure their pair-wise RTTs, and then each pair of sites averages their link's RTT value so that the final RTT map is consistent across sites. Under a fully connected network, this inter-state RTT map is stable [62], with prior work [120] showing less than 6% month-to-month difference in median latency on Azure. Through this initial RTT map, sites compute the automated zones' membership, at which point bootstrapping concludes, and Limix starts operating without any further assumptions about the timing or location of partitions.

2.6.2 Autozoning strawman

We first describe a strawman that bounds the exposure from *a particular site* to any item. We then extend this strawman into a scalable approach to bound the exposure access to any item via *any site*.



(a) **Strawman**: bounding exposure w.r.t. a *single* site.

(b) **Autozoning**: bounding exposure for accesses to any item via *any* site.

Figure 2.6: Autozoning strawman and final design. The strawman bounds exposure w.r.t. a single site at an overhead logarithmic in network width, but does not scale when applied to all sites. The final design employs compact graph approximations for scalable zoning and bounds exposure for all accesses.

To bound the exposure of accesses from *a particular site* to any object, we could simply build overlapping zones centered on that site. A naïve, non-scalable approach would be to build many concentric zones of slowly increasing radius. A more scalable design is to choose exponentially increasing zone radiuses, as depicted in Fig. 2.6a. Exponentially increasing radiuses prioritize geographically close accesses, which likely have a small RTT [5]. As RTTs increase, localities and exposure guarantees are less tight. We justify this tradeoff through the scalability of the approach, which builds a number of zones logarithmic in the network width.

This simple strawman, however, only bounds exposure for clients accessing the system through the particular site at the center of the zones. In fact, if a client chooses another site, in line with Limix's goals of supporting dynamic access, then the client does not have any exposure guarantee. We show next how to build zones that bounds exposure for any client accessing any object through any site.

2.6.3 Autozoning through compact-graph approximations

Autozoning builds on techniques from compact graph summarization theory [115, 116] to guarantee exposure for all client and all objects, while optimizing the number of created zones, hence optimizing the system overhead. Autozoning has two goals: (1) Bounding the exposure of a user accessing an item. (2) Scaling to large deployments by incurring a logarithmic load on sites. For the first goal, recall that the exposure of a client locating an item is given by the smallest zone containing both the client and the item. Our insight is to use compact graph techniques to formally guarantee an upper bound on the zone RTT diameter, hence on the user's exposure. Specifically, we want to ensure there exists a configuration

Algorithm 6: Autozoning algorithm at $\mathsf{SiteCoord_u}$.

Input : $\mathsf{SiteCoord_u}$, the site coordinator running autozoning
Sites, a set of all sites running the system
NrLevels, the number of levels for compact graph approximations
RTT, a matrix of RTT measurements between each pair of sites
Output: u.Zones, the zones that site u is part of

```
1 for v ∈ Sites do
2 |   v.Witnesses ← ComputeWitnesses(Sites, NrLevels, RTT)
3 |   if RTT[u][v] < v.Witnesses[u.Level + 1] then
4 |   |   u.Cluster ← u.Cluster ∪ v
5 for v ∈ u.Cluster do
6 |   for Radius_i ∈ i * 2^i do
7 |   |   if RTT[u][v] < Radius_i then
8 |   |   |   u.Zones[Radius_i] ← u.Zones[Radius_i] ∪ v
9 return u.Zones
```

service that has pointers to any item which is "close enough" to any client. Compact graph techniques approximate the distance between any two nodes – in our case, between a client's site and an item – to *at most* $(2 \times k-1) \times \overline{uv}$, where $\overline{uv}$ is the nodes' exact distance and k is a system parameter. They also provide the path between u and v matching this approximate distance. Autozoning groups the sites on the path between the client's site and the item into a zone. Intuitively, building such zones for all user-item pairs ensures that *any* user u looking up *any* item i is *guaranteed* to find a "small enough" common zone of diameter *at most* $(2 \times k-1) \times RTT(u,i)$.

The second goal of autozoning is to scale to large deployments. Limix relies on two techniques for scaling. The first one comes from compact graph approximations: To (recursively) compute the paths for the approximate distances between *any possible* client-site to item-site pair, each site only needs to know about $O(\log N)$ other sites, where N is the total number of sites. Even so, if we built all zones incrementally spanning sites on each path – and imposed the zone deployment load on the member sites – the cost becomes prohibitive because of the large constants in $c * O(\log N)$. Instead, we use exponentially increasing zone radiuses, as in the strawman. As a result, each site participates in and runs a logarithmic number of zones. Fig. 2.6b depicts the autozoning design, omitting the global zone for simplicity.

Zone construction. Alg. 6 depicts the zone construction. Compact-graph approximations use the sites as landmarks for approximating distances. Higher-level sites act as global landmarks to approximate large distances, whereas lower level sites act as local landmarks. Each site obtains level i with probability $N^{-i/k}$ (k is the number of levels). To approximate distances, each sites maintains a set of sites as contact points, called its *bunch*. A node u explores sites in ascending distance from itself, and adds a site v in its bunch if v's level l_v is no smaller than that of any sites explored so far (including u). The sites v closest to u at every level form u's witnesses (line 2). The inverse concept of a bunch is a *cluster*. v's cluster is the set of sites around v, which are "close enough" to know about v as a landmark (lines 3-4). Every site is a

landmark and builds zones along its cluster, using exponentially increasing RTT diameters (lines 5-8).

2.6.4 Exposure-limiting guarantees and scalability

Lamport exposure bounds. From the cluster construction, a node at level $k-1$ has all nodes in its cluster and, thus, creates a Global zone. Because node w builds zones on its cluster, and u and v are in its cluster (in other words, u and v have w in their recursive bunch), u and v are guaranteed to be in a zone of diameter at most $D = i * 2^i$, where i is the smallest such that $i * 2^i \geq (2 \times k - 1) \times RTT(u, v)$. By construction, *any two* sites u and v – alternatively, a user contacting site u to look up an item stored on v – are guaranteed to find such a zone, providing guaranteed bounds on the Lamport exposure.

Load. The size of a node's bunch is a key property determining the number of zones that a site is a member of. From the probability distribution of level assignment, we expect to accept approximately $B = \frac{1}{n^{-1/k}}$ nodes into u's bunch at each level i. Thus, each node's bunch has, with high probability, size $|Bunch_u| \approx B \times k = B \times \log_B(N)$, which upper bounds the number of zones u participates in. Factoring in the exponential zone diameter increase, u participates on expectation in a polylogarithmic number of zones, or $O(\log N)$.

2.7 Implementation

We implemented a Limix prototype of a configuration service interfacing an existing data store. Limix stores its configuration in a per-zone strongly-consistent KV store. To store the configuration, each zone uses CockroachDB [77], a widely-used strongly-consistent data store. Although CockroachDB has rich functionality, Limix only uses its basic KV store API read and compare-and-swap. A Limix coordinator runs on every site, providing an API to query items. Our implementation is written in Go, with bash scripts for test infrastructure.

Startup. Each site in Limix runs a startup script, which takes as input a list of participating sites and either jurisdictions or autozoning with the number of levels. In the case of jurisdictions, the zone membership is given. For autozoning, each script measures its RTT to all other sites, obtains each site's level, and runs locally the Alg. 6 to establish the zone membership. Then the scripts communicate in order to start a configuration store, i.e., CockroachDB instance, per zone.

Processing lookup requests. Each site runs a Limix coordinator that provides a client API itemLookup(key) : (zone, logicalTimestamp) to look up strongly-consistent data-plane items. To answer queries, each site's coordinator queries the configuration stores running on that site, through site-local configuration store connections using the Go pq driver [7] (a PostgreSQL-compatible driver). The configuration stores on each site communicate with other configuration stores within the zone, according to CockroachDB-internal implementation. A Limix

coordinator running on a site part of zone Z receives callbacks from the existing external data store when the data store creates, updates, deletes or migrates an item in that zone. For simplicity, we assume an existing external CockroachDB instance for data items. Through callbacks, the Limix coordinator updates pointers and authoritative zone information through site-local configuration store queries.

2.8 Evaluation

We evaluated Limix's resilience to network partitions and overheads. Our first experimental setup considers jurisdictions, and evaluates Limix's overhead. Our second experimental setup focuses on Limix autozoning and its resilience, and compares Limix with Physalia.

We used two testbeds for our experiments, both orchestrated using Kubernetes.

Cluster testbed. The cluster testbed runs 40 Kubernetes site in a local cluster with a simulated network, which allows for more expeerimental flexibility. Each site requests 15GiB memory and one hyperthread of an Intel Xeon Gold 6240 CPU @ 2.60GHz. The delays between sites represent real-world delays of a globally distributed topology, as follows. Using CAIDA's Archipelago (Ark) Measurement Infrastructure [5], we first selected 90 monitors of types infrastructure, research or education (Africa 8, Asia 13, Europe 26, North America 31, Oceania 4, South America 8) and selected 40 random monitors for our site locations. To compute the RTT between two monitors at geographical distance d, we averaged the two monitors' median RTT for that distance as reported by CAIDA. The minimum RTT between two different monitors is 0 and the maximum is 602.25 ms.

AWS testbed. The real-world testbed is deployed on Amazon Web Services (AWS) and spans 20 sites (US: East 4, West 4; Canada: central 2, Asia-Pacific: SouthEast 2, NorthEast 3, EU: Central 1, West 2, South 2). The server at each site has 16 GiB of DDR4 SDRAM, up to 10 Gbps bandwidth and 2vCPU on 3.1 GHz Intel Xeon processors. The minimum RTT we measured between sites is 0.43 ms and the maximum is 250.31 ms.

Software setup. Limix autozoning experiments use two levels for building zones through compact graph approximations, and zone diameters of $2(i-2)\sqrt{2}^{i}$, where $i \geq 3$. For our Physalia implementation, we use cells of up to 50 ms maximum RTT, based on the 99th percentile write latency reported in the published paper. Limix zones and Physalia cells all run Core CockroachDB ver.20.1.

Workloads. We evaluate Limix using configuration writes (W), because they are the critical operations during gray failures (Section 2.2). Specifically, we use pairs of write (W-W) operations between pairs of sites. Each W-W pair concerns the same item. The operation pair emulates a reconfiguration for that item, for example after an item migration. The reconfiguration write issued by the second node has a strong consistency dependency on the first. We use the term

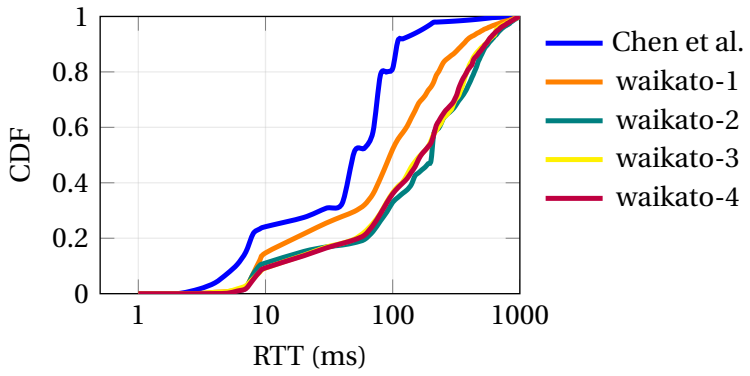


Figure 2.7: CDF of reconfiguration RTTs.

"reconfiguration RTT" to refer to the RTT between the two writers. The goal is to understand the dependencies of the reconfiguration operation to succeed, given a known prior location of the configuration. In our experiments, we run the workload as Poisson arrivals, at a rate of 20 pairs per second.

We use real-world distributions for the reconfiguration RTTs based on metropolitan traffic traces. We chose metropolitan traffic traces as they capture more local traffic, as opposed to backbone links that might miss local traffic. Chen et al. [38] provide an RTT distribution over a 10Gbps metropolitan link – henceforth called trace 1. We compared this trace to four traces of the Waikato dataset [87], whose RTT distribution we extracted using a similar methodology as [38]: we matched data packets with the respective ACKs. These traces rely on public datasets and the extraction methodology avoids raising ethical issues. Figure 2.7 shows the cumulative distribution function (CDF) for the reconfiguration RTT. Given the similarity of the distributions, and the fact that trace 1 is more recent, we used trace 1 for generating the reconfiguration RTTs.

2.8.1 Jurisdictions: availability and costs

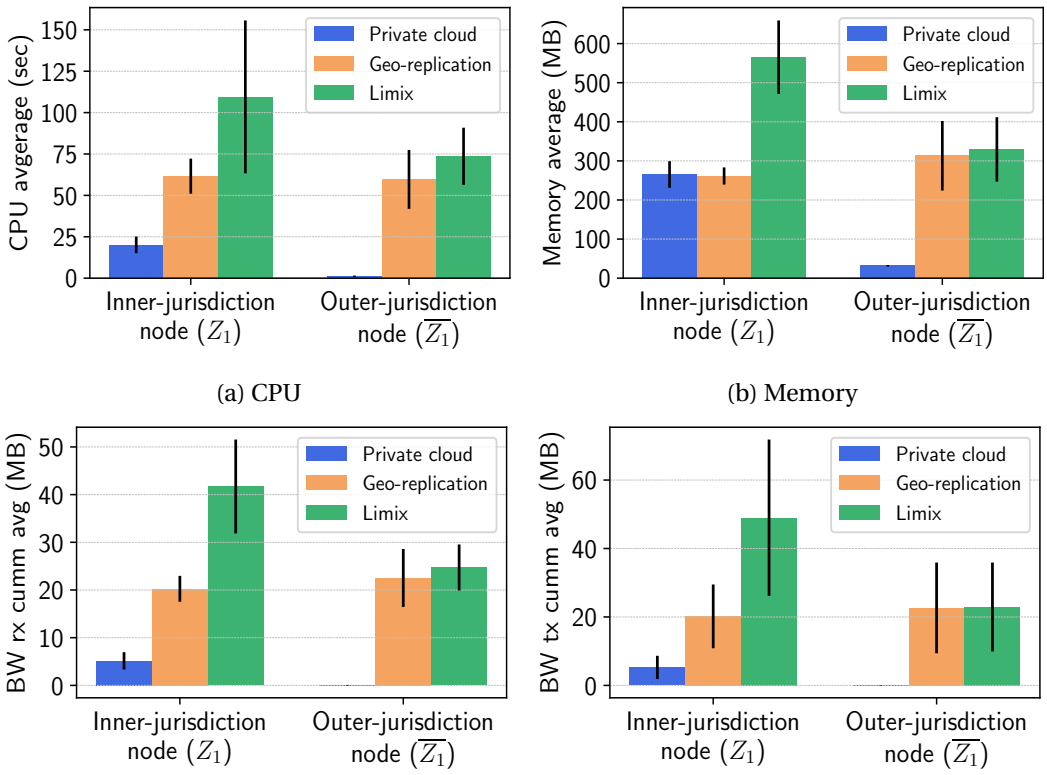
Our first experiment focuses on a simple Limix deployment in which each item exists in only one local zone, in addition to the default global zone. In this scenario we consider each of the disjoint local zones to represent an administratively-defined *jurisdiction*, such as a country. The experiment answers the question: "*What are the availability benefits and cost per Limix zone in this scenario?*"

Methodology. We ran the experiment on the cluster testbed. We focus on one particular jurisdiction centered around a CAIDA monitor in Europe West, and choose an RTT radius of 50 ms around the center, which roughly corresponds to the EU-West jurisdiction. Prior work has also shown that latencies of roughly 30ms correspond to country-wide RTTs in Europe [38].

We are interested in the overheads during no-partition conditions, and in the availability during partitions. For this experiment, we run a synthetic workload with 2000 pairs of W-W

Feature	CockroachDB		Limix
	Geo-replication	Private cloud	
Availability Z_1 reconfig.	0%	100%	100%
Global configuration mgmt.	✓	×	✓

Table 2.1: Jurisdictions: features, measured availability.



(a) CPU

(b) Memory

(c) Bandwidth RX

(d) Bandwidth TX

Figure 2.8: Jurisdictions: compute, memory, and bandwidth overhead.

operations: 1000 W-W pairs are for reconfiguration in EU-West, and 1000 for global reconfiguration. For the availability experiment, we run the same operations, but we disconnect EU-West from Global. We denote EU-West by Z_1 and Global \ EU-West by $\overline{Z_1}$. In Limix, nodes in Z_1 are part of two zones, whereas nodes in $\overline{Z_1}$ are part of one zone.

Availability. We compare Limix against two baselines: core CockroachDB with geo-replication in the Global zone – henceforth called Geo, and core CockroachDB deployed as a private cloud in EU-West only – henceforth called PCloud. Table 2.1 summarizes the features of the three designs. Geo offers global configuration management, but reconfiguration in Z_1 fails under partitions. The reason is that, under partition, Z_1 cannot reach a majority of configuration replicas. PCloud succeeds reconfigurations in Z_1 under partition, but offers no global configuration management, because all configuration is in Z_1. The experiment confirms that Limix offers both availability in Z_1 *and* global configuration management.

Overheads. For the same workload, we report the memory, CPU and bandwidth overheads under no-partition conditions. Fig. 2.8 shows these overheads for nodes in Z_1 and $\overline{Z_1}$. As expected, PCloud nodes have the lowest overheads, but PCloud lacks global manageability. PCloud nodes $\overline{Z_1}$ simply forward client requests to the closest node in Z_1, hence use almost no resources. The nodes in Z_1 running PCloud have lower overheads than Geo nodes because the PCloud deployment is smaller, and requires fewer resources for coordination between nodes, for example. For its improved resilience guarantees, Limix sites in Z_1 spend about 2x the memory, CPU and bandwidth compared to Geo. The memory overhead stems from nodes in Z_1 running two instances of CockroachDB: one corresponding to the inner- and the other to the outer-jurisdiction. To explain the CPU and bandwidth overheads, recall that, when a Limix site in Z_1 executes a write, it also writes to $\overline{Z_1}$. However, Limix sites in $\overline{Z_1}$, which do not have an improved resilience guarantee compared to Geo, have overheads very similar to Geo. We conclude that Limix is suitable for a highly configurable pay-as-you-go deployment, where every extra resource spent provides an immediate increase in availability.

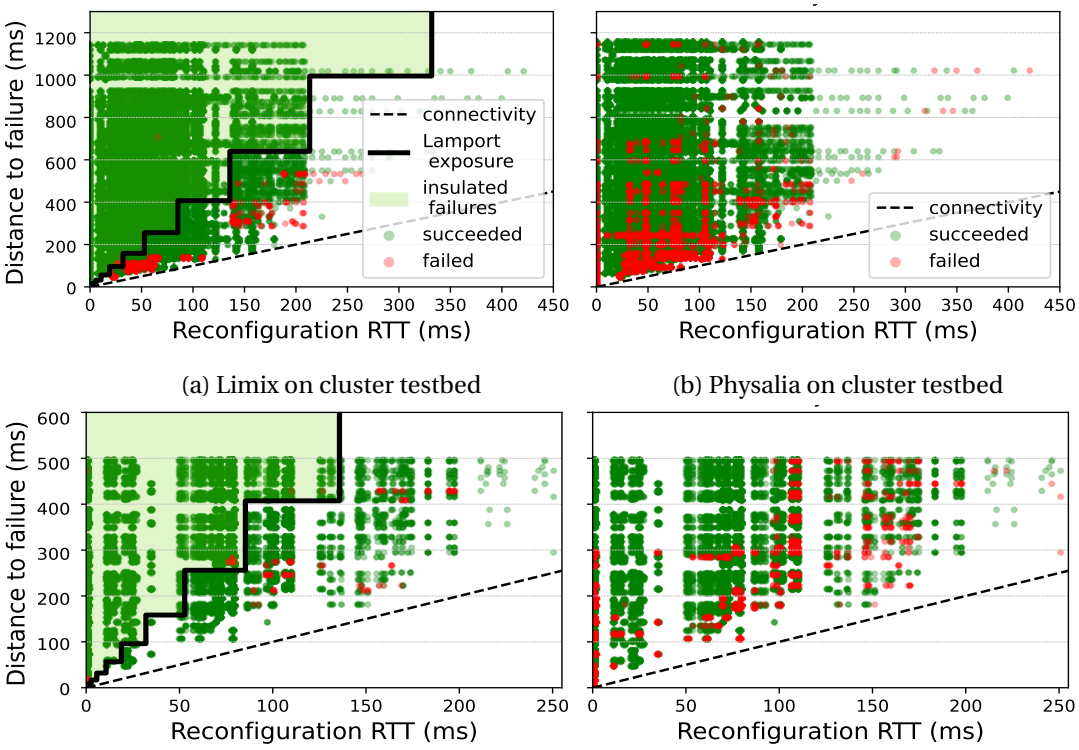
Optimization. Reducing load by "pacing out" lookup requests to zones, is a likely desirable optimization in practice, as described in Section 2.5.2. A client can send the same lookup request to one-zone-at-a-time, starting with the closest zone, only contacting other zones if the client fails to receive a reply within a given timeout. Pacing out requests likely incurs minimal overhead not only in "normal-conditions" when failures and partitions do not disrupt zones, but also for data predominantly accessed in a few close localities. We defer to future work the evaluation of this optimization.

2.8.2 Microbenchmark: availability guarantees

This experiment evaluates to what extent the configuration of localized data is exposed to remote gray failures. This time, however, no jurisdictions are given: we use autozoning and test Limix's availability guarantees in comparison to Physalia. We ask the question: "*If a random site runs a reconfiguration for a random item, to what extend could remote failures cause the reconfiguration to fail?*"

Methodology. We generate pairs of writer clients; each pair performs a configuration write at a site chosen uniformly at random. The pairs are chosen as follows. We select 30 random sites, and in a random radius around each site of up to 150ms chosen uniformly at random, we generate 1000 interacting pairs. We disconnect the network at a distance $R_1 2 = x * R_1, x = \{1,2,3,4,5\}$, and run each experiment separately. We then depict the result of the interaction (success or fail) relative to the RTT between the interacting nodes and the distance to the failure. Each pair writes a different key from the other pairs to avoid dependencies across pairs. As a result, the reconfiguration RTTs (i.e., the RTT between the two sites) are distributed uniformly at random.

We are interested in what dependencies the second writer might have on other sites, and how far in network distance these sites are: Dependencies on other sites could cause the second



(a) Limix on cluster testbed

(b) Physalia on cluster testbed

(c) Limix on AWS

(d) Physalia on AWS

Figure 2.9: Comparison of availability of Limix and Physalia for different failure scenarios and testbeds.

writer's configuration write to fail, e.g., by triggering a correlated failure. For this purpose, we select an area of a random radius R around the second writer, and we partition the network along the zone's border. The two writers are never partitioned from each other, because by design the second writer has a strong consistency dependency on the first, and then the second write would fail. By running this experiment, we test whether the second writer has dependencies outside the partitioned area.

Figure 2.9a, Fig. 2.9b depict the success or failure result for each writer-writer pair. The x axis represents the reconfiguration RTT, and the y axis represents the network distance to the partitions. Thanks to Limix's exposure guarantees, reconfigurations in Limix succeed more frequently than in Physalia. Both system register failures below Limix's shield, showing that they do have dependencies nearby. However, Physalia sites also fail when failures are relatively far: Physalia records failures above Limix's shield because it provides no guarantees for all the pairs interacting across cells. Even if two sites are relatively far from failures, if they are in different cells and their cells are partitioned, their interaction might and do fail. In contrast, Limix autozoning provides a clear availability guarantee, applicable to all interacting pairs: When failures are farther than the Lamport exposure bound of the two writers, depicted using the black shield line, reconfiguration is guaranteed to succeed.

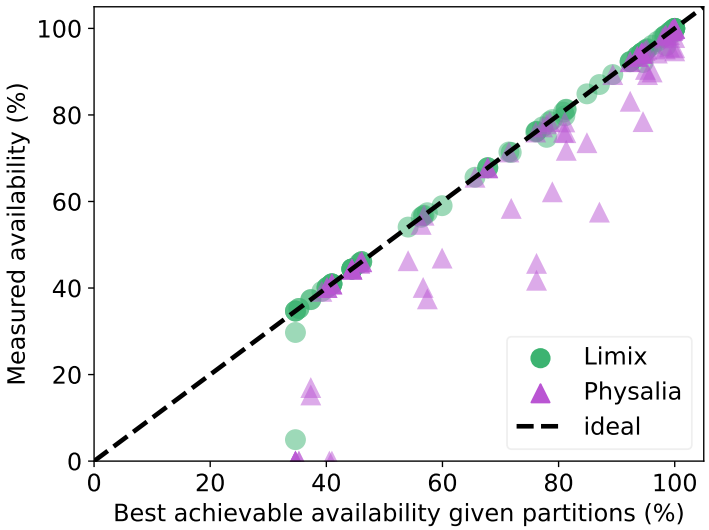


Figure 2.10: Comparison of Limix to Physalia on AWS and realistic workloads.

2.8.3 Availability under real scenarios

This experiment, like the previous one, tests to what extent the configuration of localized data is exposed to remote gray failures, but on a real network and using realistic trace-based data. Because the AWS testbed has lower RTTs that are more clustered, Physalia cells are mostly between 10-30ms RTT diameter, with a single cell up to 50ms. Our methodology is similar to the experiment above, with the only exception that the workload of 1000 reconfiguration pairs of each experiment has a distribution of reconfiguration RTTs matching the one in trace 1 (Section 2.8). The workload is global, thus some interacting pairs cross the partition boundary.

Figure 2.9c, Fig. 2.9d plot the results only for the non-partitioned interactions. Our results generally mirror the ones we obtained on the cluster testbed, with Limix outperforming Physalia in almost all tested cases, and for the same reasons. There are, however, two notable differences from the previous experiment. First, Limix registers a few failures close to but above the shield. These failures correspond to a some sites with a more unstable RTT than the others, whereas we depict the RTT measured at bootstrap time. However, the difference in RTT was minor, and for all the other sites we observed no violation. Second, we observe a more pronounced clustering effect of the plotted points, matching roughly our more clustered topology.

We also summarized the results for each tested workload as percentage of successes of the maximum possible availability for the non-partitioned pairs. Fig. 2.10 shows that Limix's success rate is close to 100% in most cases, and significantly outperforms Physalia. We conclude that Limix provides strong guarantees on a variety of testbeds and workloads.

2.9 Related Work

CAP tradeoffs. Faced with partitions, some systems choose to relax consistency in favor of availability. Gemini [80] distinguishes access types that require a strongly- or eventually-consistent reply; this technique is known as segmentation [57]. Dynamo, a highly-available data store, takes a similar approach [44]. Seredinschi et al. [60] provide the user with several replies, increasing in consistency guarantees, enabling the client to perform speculative work. In Limix, all accesses are strongly consistent.

Availability during failures. Several strongly consistent systems employ replication to survive failures, however, they assume *uncorrelated failures* across sites [79, 94, 120, 123, 129]. But failures might not be independent across a wide area, for several reasons. For example, machines across sites run the same software and are vulnerable to the same bugs [22, 61]; nodes' hard disks fill up at the same rate [22]; short-lived and, less frequently, long-lived (partial) partitions separate sites from each other, causing a domino of failures and ultimately unavailability [11].

Glacier [61] employs massive replication of data to minimize the probability of data loss during large-scale "massive correlated failures". As opposed to Limix, however, Glacier considers data stores with immutable objects, whereas Limix targets strongly consistent data stores with editable objects. Glacier optimizes storage and reconfiguration to survive Byzantine failures, whereas Limix design a metadata service that is immune to remote failures and slowdowns.

Availability metrics. Unlike the availability metric that Hauer et al. [64] recently proposed, *windowed user-uptime*, which *reactively* analyzes failures after they occur, Limix *proactively* limits exposure in the first place. Limix's Lamport exposure concept ensures worst-case guarantees for a user, including rare events that might not significantly affect the median availability, but still account for many hours of downtime. In contrast with the *blast radius* notion proposed by Brooker et al. [22], which attempts to reduce the damage caused by a partition, Limix focuses on users, aiming to insulate their accesses from *any* partitions or slowdowns outside a relevant local zone.

Traffic engineering. Traffic engineering techniques are orthogonal to Limix's application-level exposure limitation, and could be used in conjunction with Limix. Techniques such as IP Anycast [8] and geolocation [72] or GeoDNS [56], Oasis [55] and ClosestNode [127] connect a client to the physically closest server. Donar [126] maps clients to replica server based on client-to-replica latency and server load. Recent work in traffic engineering [103, 121, 130] incorporates several application performance metrics into routing, not only for client-to-replica mapping, but also between the replicas.

Zoning. Limix's autozoning protocol builds on landmark [119] and compact routing schemes [115, 116] that store less routing information in exchange for longer routes. Other zoning approaches are possible, however. Awerbuch et al. [17] propose clustering techniques that preserve locality in arbitrary networks that could be used for Limix zoning. In a more recent work, Trajanovski et al. [117] consider the problem of correlated failures that manifest simultaneously in different geographic regions of a provider's network. They propose algorithms for finding a critical region in the network – a planar geometric region that creates the highest network disruption if the nodes in that region fail. The authors also propose a heuristic for building regions that decrease the impact of a regional failure by ensuring that any two nodes can communicate through two paths that do not cross the same region.

Scalable Overlay Networks. Scalable overlay networks, such as distributed hash tables (DHTs) Chord [108], Pastry [101], Tapestry [133] efficiently locate items by implementing a lookup operation. Although the goal to look up an item is similar to Limix, these schemes are different in other provisions. A major difference is that DHTs do not aim to collocate metadata with the data, which may lead to metadata being stored far away from users or in jurisdictions that contradict regulation, for example. Some DHT schemes, such as Kademlia [85] and Tapestry [133], include provisions attempting to improve locality. From a locality viewpoint, SkipNet [63] is an overlay network that stands out, because it explicitly design for locality. It takes organizational boundaries into account when placing contents, and ensures that traffic between nodes in the same organization stays within the organization. DHT schemes, however, do not have built-in support for item strong consistency, in the sense that changes to an item create a new unrelated item. In contrast, Limix supports strongly consistent item lookup in data stores with editable items.

2.10 Summary

We proposed Limix, the first distributed metadata coordination service that enables global management while protecting local accesses from distant failures. Limix designs a control plane that limits exposure by running a localized coordination service per zone so that, if some larger zones become partitioned and unavailable, local zones can continue functioning. Limix's control plane supports existing strongly consistent key-value data planes with item migration. Limix also defines several exposure boundaries, and an efficient and scalable autozoning algorithm with tight exposure bounds and polylogarithmic load overhead. We provide a prototype implementation of Limix on top of the key-value store functionality of CockroachDB. Limix routinely outperforms Physalia by as much as 30% better availability, on synthetic but realistic workloads, as shown by our wide-area experiments on AWS.

3 A Trust-but-Verify Lamport Exposure-Limiting Blockchain Architecture

Wide-area distributed ledger (or blockchain) deployments run wide-area consensus to agree on a total ordering of transactions. By doing so, blockchains prevent double spending, because only the first transaction that successfully spends a certain token or asset is valid. Later transactions attempting to spend the same token are not valid.

However, wide-area consensus subjects transactions to a wide-area Lamport exposure (defined in Section 2.6.1), possibly disrupting availability or safety. Ledger designs based on classical Byzantine Fault Tolerant (BFT) consensus assume that a sufficient number of validators are alive and reachable [75, 76, 131]. Ledgers based on proof of work assume that partitions are unlikely to last "too long" [49, 92, 128]. Yet, reality may not conform to these assumptions. Partitions and failures may prevent or delay a quorum of validators from communicating. Because public blockchains use the Internet as the communication medium, denial-of-service (DoS) attacks can cut communication by exploiting network sharing effects [109, 111]. An attacker could even selectively censor packets through routing attacks that redirect traffic via an attacker-controlled autonomous system (AS) [14]. Even absent such attacks, ASes forwarding blockchain traffic can selectively discriminate traffic, perhaps for economic gain, by adding delays [54]. Such partitions may prevent or delay a quorum of validators from communicating, causing either a loss of availability (liveness) in classical BFT consensus, or forks (loss of consistency) in proof-of-work blockchains.

Should transactions that transact a global asset in a localized setting incur the penalties of wide-area consensus? Consider a scenario where Alice purchases daily coffee from the local coffee shop; Bob the merchant may or may not require world-wide consensus to accept the payment. We believe that Alice and Bob should be able to choose: If the merchant and the buyer already know each other or have some other established trust relationship, then the merchant might be willing and even prefer it to accept the transaction with a local consensus, for example within Switzerland. Local consensus exposes the transaction to merely local failures or slowdowns, which are a fraction of the global failures or slowdowns (Section 1.2.2-Fig. 1.1). Notably, even in a total shutdown World War Three scenario where this locality is disconnected from the network, but is connected enough locally, e.g., via opportunistic

networks [124], local transactions can still be live in a classical BFT consensus blockchain. The wide-area consensus on payment confirmation will arrive later, with a higher latency, and would still expose any double-spending to the merchant.

We introduce Nyle, a blockchain architecture that limits transaction Lamport exposure. Our system has three advantages: (1) Nyle survives disaster scenarios where the global network is partitioned and cannot reach global consensus, thanks to its trust-but-verify architecture that capitalizes on existing trust relationships between the transaction parties; (2) Provided a "know your customer" (KYC) system in place, Nyle enables a blockchain framework based on jurisdictions that may help enforce regulation; (3) Nyle systematically limits transaction exposure and can improve transaction confirmation latency when using latency as the Lamport exposure metric.

A locality-aware blockchain like Nyle that takes user trust preferences into account can have a significant impact. First, payments have a strong locality property with respect to geography, e.g., a recent McKinsey global payments report shows there are at least twice as many intra-border payments than cross payments [41]. Second, other systems already explore flexible quorums based on user trust [27, 82]. In contrast to these systems, however, Nyle not only empowers users with quorum options, but it does so in a scalable, structured locality-aware manner, avoiding Stellar's unsafe configurations [82] and obtaining a significant impact.

Thanks to its support for various Lamport exposure metrics, Nyle offers a systematic method for shielding transactions against a variety of risks. For example, in Swift, the global standard for payment and securities trade transactions, there is no simple way to bypass specific corresponding banks from certain international payments. Whereas in the current ecosystem, there is no mechanism in place that offers users meaningful tools to reduce exposure (Section 3.8). The same situation as in Swift may occur with layer 2 payment channels, where it may be difficult to eliminate payment hubs between transacting parties. Asymmetric trust approaches such as Stellar [82] enable choice of quorums but without much guidance, which can create risky configurations. In contrast, Nyle's zone definitions can account for such constraints, guaranteeing that only validators within a zone are able to influence the liveness of intra-zone transactions.

Another orthogonal benefit of exposure-limiting blockchains is the potential to enforce jurisdictional legal frameworks for intra- and inter-jurisdiction transactions [118], such as the travel rule [110]. Transactions in Switzerland, for example, could be subject to financial regulations in use where the security changes hands, i.e., in Switzerland. These regulations hold both for Swiss-sovereign tokens, e.g., Swiss francs, which are valid globally but issued by Switzerland, and for global non-Swiss issued tokens, such as US dollars. Assuming a KYC system is already in place for blockchain users and validators [40, 97], Nyle can naturally implement such jurisdictional legal frameworks.

This chapter describes Nyle and argues that it meets the aforementioned objectives. We begin by providing the background for this work in terms of a typical distributed ledger design.

We then proceed with the system model, assumptions, and objectives. Then, we provide an overview of our approach and identify the challenges we must tackle. Next, we describe the design of Nyle's secure zoning algorithm and the trust-but-verify architecture that enables us to reduce transaction Lamport exposure. We provide information on the intra- and inter-zone transaction systems and their security arguments. Next we provide a simulation, showing that Nyle significantly reduces exposure compared to Omniledger, with 60% of transactions witnessing 70% less exposure in Nyle, at a cost of 10x increase in load. We end by discussing related work and peripheral issues that fall outside the scope of this work.

3.1 Background

We present a brief review of distributed ledgers, their key participants, the conventional trust assumptions, and the qualities ledgers typically strive to assure.

A blockchain is a peer-to-peer system run by nodes connected through a network such as the Internet. When the nodes are run by different organizations or individuals, the blockchain is said to be decentralized. These nodes build and maintain the blockchain, which is a data structure containing a totally ordered collection of transaction blocks. Blockchains have two important properties: (i) They are append-only, meaning that the blockchain length strictly increases by having nodes add new blocks at the end of the blockchain; and (ii) They are immutable, meaning that the contents of blocks cannot be changed: once added, blocks are read-only.

Blockchain systems have two types of participants: nodes, as mentioned above (also called validators), and clients that issue transactions. A client that issues a transaction, for example an asset transfer, signs the transaction with his private key, and send the signed transaction to one or more validators. Each node that receives a transaction validates it by checking the signature with the issuer's public key, and also checking that the issuer is in the possession of the asset. The validation fails if the issuer client is not the asset's owner, for example because he already spent the asset in a previous transaction, or because he never owned the asset. The node then runs consensus with the other nodes to agree on the transaction's order in the blockchain, then create a block and add the new transaction to the blockchain. Each block also contains the hash of the previous block to enforce the block ordering. Finally, the nodes store the new block.

Blockchain architectures use various consensus approaches, which impact their assumptions and transaction guarantees. Broadly speaking, consensus protocols aim for safety and liveness. Safety means that "nothing bad ever happens". For blockchains, a safe consensus protocol ensures that all live, honest nodes agree on the transaction ordering, a transaction's outcome and, once a transaction is agreed on, it cannot be altered or revoked – often referred to as the transaction being final. Because no two honest nodes agree on different transactions for the same blockchain index, safe blockchains prevent double-spending by design. Liveness means that "something good eventually happens", which in blockchain terms refer to the blockchain

eventually accumulating transactions. For example, an always idle consensus would be safe, but not live.

Byzantine nodes can by definition deviate arbitrarily from the protocol, for example by validating double-spending, equivocating or being silent. Classical Byzantine fault tolerant consensus algorithms [35] are safe as long as less than $1/3$ of the nodes are Byzantine, but ensure liveness only when the message propagation delay does now grow indefinitely. Otherwise, the algorithm is safe but may block. For example, the algorithms block when there is not a quorum of nodes that agree on the same value, either because Byzantine nodes equivocate, or are silent, or because there are network partitions that prevent a quorum of nodes from communicating.

Proof-of-work consensus algorithms are always live, because by construction they periodically append new blocks. However, they are safe only if more than $1/2$ of the computational capacity is in the control of honest nodes. Because of network asynchrony, the network may not propagate a block to all nodes before the next block creation time. Thus, honest nodes that do not observe new blocks may fork the chain by creating their own new blocks. When the network is synchronous again, the nodes need to decide which of the diverging chains to adopt, and PoW protocols typically adopt the longest chain. All honest nodes adopt the longest chain and, because they have more of the computational capacity, other forks can never become longer and overtake the main chain.

Similarly to Limix (Section 2.6.1), we define the Lamport exposure of a transaction as the set of validators and network links between validators that may cause the transaction to fail or be slow. In particular, the set includes all validators that forward a transaction, and execute consensus on a transaction.

3.2 Setting and Goals

This section describes the Nyle's setting and goals. We first describe the system model, outlining the fact that Nyle works with existing blockchain systems, which it enhances with exposure bounds. Next we describe the attacker model and complete the section with goals.

3.2.1 System model

Consensus. Nyle enhances existing blockchain architectures with exposure-limiting properties inspired from Limix. Nyle treats the blockchain consensus algorithm as an abstract black box. For now, Nyle focuses on classical BFT consensus algorithms, such as PBFT [34], HotStuff [131], Byzcoin [75], and defers other consensus algorithms such as proof-of-work or proof-of-stake to future work.

Validators. The literature defines validators as the nodes that run consensus to maintain the blockchain, validate transactions, store blocks. Generally speaking, BFT assumes a closed, known set of nodes that run consensus. However, ByzCoin showed it is possible to open up the consensus group in PBFT through a proof-of-work membership algorithm ensuring that an attacker with a percentage x of the computational power is not expected to register much more than the same percent x of validators. The result is that, in any given time interval, there is a closed, known set of validator nodes.

Network. We consider the set of validator nodes to be interconnected through a point-to-point wide-area network. In contrast to the datacenter-oriented system model of Limix, Nyle is intended to work with hundreds of nodes connected by a peer-to-peer network. Nodes communicate by sending point-to-point messages.

Nyle adopts the network model of the underlying blockchain architecture. Several BFT-consensus blockchains make timing assumptions. One of these is partial synchrony [45], in which there exists a bound on message delivery delay Δ guaranteed to hold after some unknown time, called the global stabilization time. Another is weak synchrony [45], where Δ varies, but it cannot grow faster than a polynomial function of time [34]. With this assumption, BFT-based blockchains can bypass the FLP impossibility result [51] and guarantee liveness.

Nyle also supports asynchronous communication when the underlying blockchain supports it. This means that messages sent are eventually received, but it may take arbitrarily long until they are received. This asynchronous model has the advantage that it captures an attacker that delays messages in the network, for example with the goal of launching a DDoS attack. Deterministic consensus algorithms cannot achieve both safety and liveness in the asynchronous communication model [51]; only randomized consensus algorithms can.

Transactions and validation (confirmation). Nyle assumes a simplified UTXO (unspent transaction output) model [92] for blockchain transactions, because in this model it is simple to reason about transaction dependencies, and thus parallelize independent transactions. This model is ubiquitous in the literature, being first introduced by Bitcoin [92] and used by many existing blockchains. The UTXO model resembles real-world money or coin transfers where a UTXO is a coin, and a single transaction can take as input or spend multiple UTXOs. However, unlike in the real-world, UTXOs can be spent only once: A transaction “fully spends” the input UTXOs and creates new output UTXOs. Also, as opposed to real-world coins, UTXOs can exist in any denomination.

Nyle’s simplifying assumption is that each UTXO has a designated receiver - identified by a public key, as in Bitcoin. Nyle’s UTXO model does not support “pay to script” transactions, which are out of scope for this work.

To validate a transaction, validators need to crawl the blockchain and assess whether all input UTXOs for the transaction have not been spent in previous transactions. This means that a

transaction depends only on the transactions that outputted the input UTXOs, and possibly transactions that already spent those UTXOs. We call these *causally related transactions*. Formally, transactions T and T' are causally related if there exists a predecessor/successor relationship between T and T', perhaps transitively through other transactions. Transactions that are not causally related are independent from each other. Throughout the paper, we use the terms transaction *validation* and transaction *confirmation* interchangeably.

Each validator node stores the cryptographic material required to authenticate its result of validating a transaction. BFT protocols deem a transaction valid if a quorum of nodes verifying the transaction indicate a positive result. Each validator maintains a pair of public and private keys needed to sign transactions. BFT protocols may employ multiple digital signature algorithms. For example PBFT uses RSA digital signatures, HotStuff and ByzCoin employ aggregated signature methods (Schnorr signatures or BLS) in which nodes issue so-called partial signatures that may be aggregated to minimize message size. A threshold of aggregated partial signatures constitutes a valid full signature. We assume that relevant cryptographic assumptions hold, e.g., for the signature schemes used by the underlying consensus protocol.

Transaction Lamport exposure. Nyle operates at the granularity of zones, as Limix does, to limit exposure. In Nyle, each zone consists of a validator set that maintains a zone blockchain. Zones can be disjoint or overlap totally or partially in their validator sets.

A transaction validated by a zone that *already has the input UTXO* on its blockchain could only be exposed to the failure of validators and network connectivity within that zone. Depending on the consensus algorithm running in a zone, however, not all validators in the zone may affect transaction validation equally. For example, in leaderless consensus algorithms, a minority of slow, crashed or compromised nodes do not affect validation. Whereas, in leader-based consensus algorithms, validation incurs a performance penalty when the leader is slow, crashed or compromised. Regardless of the consensus algorithm, however, validators outside a zone have no influence on transaction validations within the zone.

If the zone does not have the transaction's input UTXO on its blockchain, the transaction exposure also includes the exposure of fetching the UTXO, as we explain in Section 3.4.3.

3.2.2 Attacker model

In contrast to Limix, which assumes fail-stop node failures, Nyle works with a Byzantine adversary. Any entity in Nyle can be compromised: transaction senders and receivers, validator nodes and zones.

Nyle makes the classical BFT consensus assumption that *globally* at most 1/3 of the validators in Nyle are compromised (Byzantine); otherwise consensus is impossible [51]. Byzantine validators can deviate arbitrarily from the protocol, for example by validating double-spending, equivocating, rewriting blockchain history or being silent. Also, Byzantine validators can be

under the control of a single adversary that learns the nodes' whole internal state, including any cryptographic keys they might hold. The attacker also controls the network point-to-point links of the compromised validators and can drop, delay, modify and insert packets on these links.

Nyle supports an adaptive adversary that can dynamically compromise nodes at any time, by adding them to the compromised node set, but he can never remove nodes, and remains subject to the total limit assumption of 1/3. We do not make any assumption on the location of the honest or Byzantine validators. In particular, the attacker could focus its power in just a few zones, with the goal of controlling more than 1/3 of the nodes in each of those zones and rewriting blockchain history in any way, including allowing double-spending, for example. We call such zones *compromised zones*. The attacker might double-spend in compromised zones, but also might just attack them via denial-of-service to grind them to a halt.

Prior work avoids attackers taking over zones by selecting zone validators at random from a validator pool with a threshold $f < 1/4$ (Section 3.8). Nyle, however, cannot use the same security through randomness techniques, because zone validators are selected based on their location matching the zone. With known zone selection policies, an attacker can focus its efforts within a certain zone and gain local control, despite globally not controlling more than 1/3 of the nodes.

To build zones securely, Nyle assume that (honest and compromised) nodes "know" their own locations. In particular, an honest node makes accurate location claims. For example, if we use geographic location as exposure metric, we assume that honest nodes cannot be under GPS attack, otherwise Nyle considers them as "compromised" nodes, counting towards the attacker's maximum global threshold of 1/3. In addition, nodes need to provide a verifiable proof for their location, for example by using VerLoc [74]. Proofs are only needed for leader-based consensus algorithms, but not for leaderless ones, as we explain in Section 3.4.1.

3.2.3 Goals

Nyle has six main goals:

- Resilience to remote network partitions and slowdowns: intra-zone transactions commit even if the zone is disconnected from the rest of the network.
- Joint sender and receiver control over which zones to trust for confirming their transaction. Nyle informs the sender's and receiver's choice by building zones with varying exposures. Smaller zones have a smaller exposure, but may be under attacker control, whereas larger zones up to the global zone are more secure, but also have a higher exposure.
- Enforcement of jurisdictional legal frameworks for intra-jurisdiction transactions, assuming a system in place that performs KYC for validators joining zones.

- No double spending through a trust-but-verify architecture. Eventually the safe Global zone validates all transactions.
- Load scaling for validators across zones. When Nyle uses Limix's zone construction inspired by graph approximation techniques, Nyle obtains similar scalability properties.
- Multi-input multi-output transactions. A transaction can spend multiple inputs, possibly each owned by a different party, and produce multiple outputs, each under the ownership of possibly different receivers.

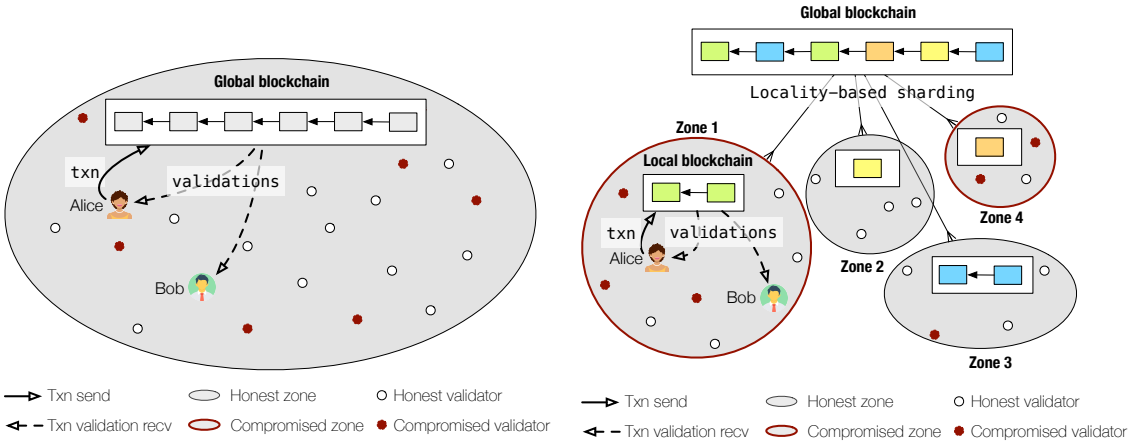
3.3 Overview

This section gives an overview of Nyle, which is an exposure-limiting blockchain architecture. We first describe two extremes of the design space that are either insecure or do not limit Lamport exposure. Next, we introduce Nyle's philosophy of minimizing the time when transactions are under attack: Nyle combines the two design extremes in a trust-but-verify approach. Finally, we outline the challenges that Nyle needs to address.

Nyle's main goal is to limit the Lamport exposure of transactions. In the design space, one extreme is a single, global zone, as depicted in Fig. 3.1a. This global zone runs the blockchain and is secure, because the attacker cannot control more than 1/3 of the nodes. However, this global zone fails to reach our goal: all transactions are exposed to global failures and slowdowns, no matter how close the sender and receiver are from each other.

The other design extreme uses many non-overlapping local zones (by some locality metric), as depicted in Fig. 3.1b, with each zone running an independent blockchain. The advantage is that such local zones shield intra-zone transactions from failures and slowdowns outside the zone. But, because our attacker is adaptive, this design is insecure: The attacker can focus its power in one or a few zones of its choice, with the goal of compromising a zone by controlling more than 1/3 of the nodes in a zone. As long as globally the attacker does not control more than 1/3 of the nodes, the attack is valid under our attacker model. Transactions in such compromised zones are never secure. Because the resulting blockchain is as secure as its weakest link, this locality-based sharding design point is insecure.

Nyle combines the local and global zones from the design points above in a *trust but verify* architecture, depicted in Fig. 3.2. Each zone's validators run an independent blockchain, just as before, and the zone's validators verify transactions sent to that zone in parallel and independently from other zones. In contrast to sharded blockchains, the novelty is that Nyle zones can and do overlap, and Nyle verifies each transaction multiple times, by local and increasingly more global zones. The sender and receiver receive multiple confirmations for their transaction. Smaller, local zones, bound the Lamport exposure of intra-zone transactions to intra-zone failures and slowdowns, and respond quickly. However, the local zone could be compromised. The slower Global zone is always secure and verifies all transactions, alerting interested clients of potential double-spending. Thus, local chains acts as opportunistic



(a) Secure global blockchain, transactions are secure but have global Lamport exposure.

(b) Non-overlapping zones with zone-local blockchains. Transactions involving compromised zones are never securely finalized.

Figure 3.1: Design space extremes for Nyle.

finalizers that parties can choose to trust in certain scenarios, and the global chain always verifies the transactions later and acts as a secure finalizer.

We assume that the global blockchain has the capacity (transaction throughput) to verify all the transactions that ever occur throughout the system, even if it does not necessarily verify them particularly quickly. For example, the global zone may use sharding as defined in Omniledger [76] and is adequately provisioned for this purpose, although the details are beyond the level of abstraction this work deals with and thus out of immediate scope. Or, Nyle may incorporate some form of rollups [47] to reduce the amount of work the global zone has to do, by allowing the global zone to verify only summaries of local-zone transaction histories rather than entire local histories - but this is an area left for future work and again not in the immediate scope of this work.

Nyle has the benefit of giving senders and receivers discretion over transaction exposure. The transacting parties can move forward with only local validations and then cross-check these validations against the Global zone validation, if they have sufficient mutual trust or have a local zone that both parties trust. This trusted local zone has a lower exposure and may permit faster progress when the Global zone validation stalls, such as when opportunistic networks or attacks cut off the local zone from the outside world or when the Global zone is simply slow.

Fig. 3.2 depicts the validation process for Alice's transaction when buying a coffee from Bob the merchant. Alice submits the transaction to one or more nearby validators, which in turn broadcast it to all validators. The transaction is verified independently by all zones that cover Alice's location, in this case by local Zone 1 and by the Global zone. Both Alice and Bob eventually receive confirmations from Zone 1 and the Global zone. If the Global zone confirmations are slow, because of temporary connectivity issues somewhere in the Global

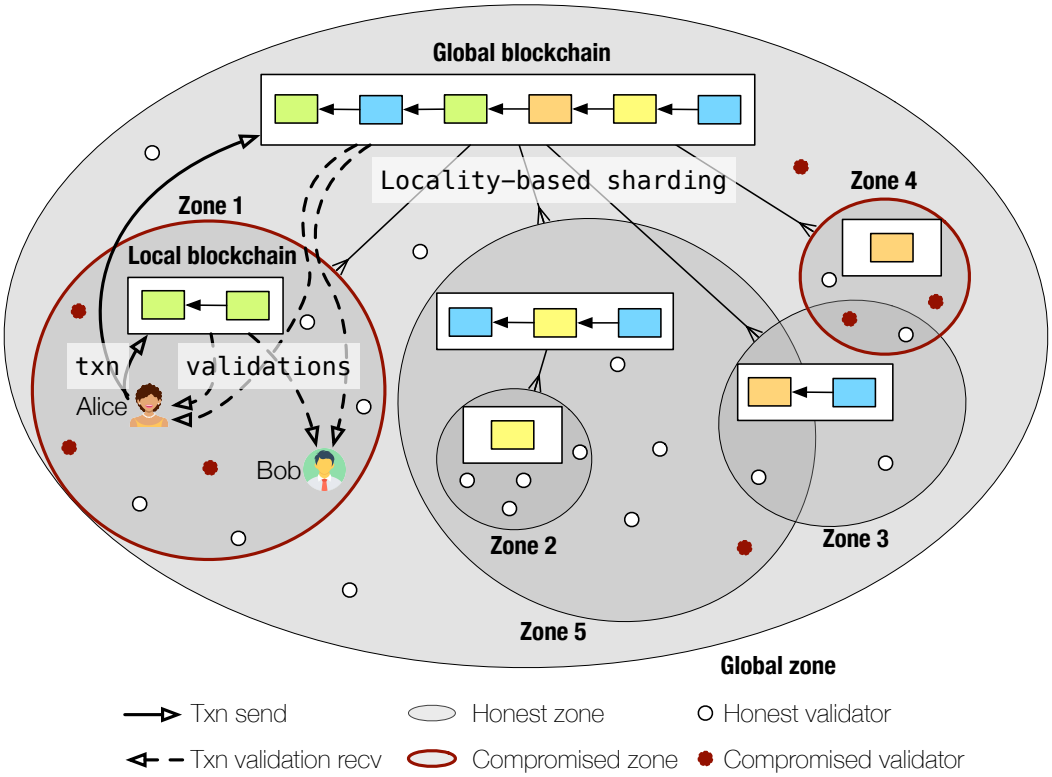


Figure 3.2: Final Nyle design using a trust-but-verify principle. Multiple zones independently validate each transaction, and the sender/receiver can choose which validations to wait for. Local zones limit Lamport exposure but might be malicious, whereas the Global zone always acts as the possibly slow but secure transaction finalizer.

zone that slow down consensus, then Bob can choose to trust the quick transaction validation from Zone 1 if Alice is a regular, trustworthy customer, and verify the transaction later using the Global zone validation, which is always secure. Or, Bob can alternatively choose to wait for the validation from the Global zone. Either way, the receiver has the choice to enable local transactions to proceed even when the zone is disconnected from the rest of the world, and verify these transactions later with the secure Global zone.

Challenge: secure zoning. There are several challenges that Nyle must overcome that are not captured by the aforementioned scenario. The first group of challenges relates to secure zoning and presents two problems. (1) The objective of Nyle is to isolate transactions from distant failures and slowdowns. What location metrics are relevant for achieving this objective? (2) Nyle insulates transactions at the zone level and constructs zones according to the locality metrics described above. How can we safely compute zone memberships that implement these metrics while validators may lie about their measurements?

Challenge: overlapping zones. Depending on zone overlap, Nyle may validate each transaction many times. As a result, many zones in Nyle contain records of the same output UTXO, resulting in three challenges. (1) How can Nyle ensure that honest zones, which run independently, do not unintentionally approve the spending of the same UTXO in different transactions, hence breaching consistency? (2) Some zoning algorithms may generate non-strict zone hierarchies. Baarle, for instance, is a border town located in both Belgium and the Netherlands, causing zones to partially overlap in a country-based zoning scheme. Limix's autozoning scheme can similarly generate non-strict hierarchies (Section 2.6.3). What effect do these hierarchies have on which zones validate a particular transaction? (3) A sender may wish to spend funds from many zones in a single transaction. How does Nyle guarantee the atomicity of transactions across zones?

Challenge: guarantees during attacks The final set of challenges relates to the security of Nyle transactions in compromised zones and with compromised users. There are three problems. (1) Local zones in Nyle are susceptible to compromise. How does Nyle manage compromised zones that violate safety, such as those that permit a malicious sender to double-spend, rewrite blockchain history, or accept wrong transactions? (2) Compromised zones could break liveness, e.g., refusing to confirm a sender's transaction, effectively censoring that sender. How does Nyle ensure that such a zone will not permanently freeze a user's funds? (3) The sender and/or the receiver of a transaction could also collude with a compromised zone, with the goal of tricking honest zones and polluting them with fraudulent UTXOs. How does Nyle counter such attacks?

We address these challenges in the following sections.

3.4 Architecture

This section describes the architecture of Nyle. To insulate transactions from remote failures and slowdowns, Nyle creates multiple zones of different Lamport exposures, each maintaining an independent blockchain. Any zone except for the Global zone could be compromised, thus Nyle builds a *trust-but-verify* architecture that enables the sender and receiver to choose which zones to trust for transaction confirmation. We focus on the two components of Nyle's architecture, addressing some of the challenges highlighted in the prior section: secure zone construction and the basics of transaction verification via the trust-but-verify architecture.

3.4.1 Secure zone construction

Each Nyle zone, specifically the validators in that zone, maintain a blockchain by running consensus to agree on transaction validity, and on transaction ordering within the zone. Each zone runs some variant of BFT consensus (Section 3.2.1), which relies on the agreement of a quorum of the validators. To form quorums, the validators in each zone need to know the

zone's membership.

The zone membership in Nyle has to fulfill five conditions:

1. The Global zone is secure, in other words less than 1/3 of the validators are compromised. Nyle ensures this condition by adopting an existing BFT blockchain membership protocol for the Global zone (Section 3.2.1).

2. The validators in any local zone are a subset of the Global zone validators. Nyle ensures this condition by distributing into local zones only validators that are members of the Global zone. Mathematically, $M_i \in M_G, \forall M_i$, where M_i is the validator set of $\mathsf{Zone_i}$, and G is the Global zone. We address this requirement in the secure zoning paragraph below.

3. Local zones can be compromised, however, the attacker remains subject to the global 1/3 limit on total number of validators compromised. In other words, the attacker can distribute the compromised global validators in zones as it wishes.

4. A validator cannot be part of two zones' validator sets unless the two zones overlap in their boundaries: $M_i \cap M_j \neq \phi$ iff $\mathsf{Zone_i}$ and $\mathsf{Zone_j}$ overlap. We address this requirement in the secure zoning paragraph below.

5. The honest zones, where the attacker controls less than 1/3 of the zone validators, guarantee their Lamport exposure bounds. We address this requirement in the secure zoning paragraph below.

Membership attacks. If the membership protocol of the Global zone does not have support for a notion of location for the validators, then an attacker may much easier compromise local zones. *Attack 1:* First, the attacker could exploit the lack of locality information and attempt to register the same compromised validator as a member of disjoint zones. Note that globally, the validator still counts as one and the attacker does not break condition 3. But, the same compromised validator would count as two validators in disjoint zones, breaking condition 4 above, and would illegitimately increase the attacker quota of compromised validators in local zones.

Attack 2: Second, the attacker could attempt to craft the location information of *honest nodes*, confusing the exposure-limiting zoning algorithm Section 2.6.3. The zoning algorithm may then incorrectly assign honest validators to honest zones. For example, an honest validator located in Germany is incorrectly assigned to the US zone. Such an attack effectively increases the Lamport exposure of the honest US zone beyond the desired zone boundaries, breaking condition 5 above.

Attack 3: The attacker could craft a *compromised* validator's location so that the validator is assigned to an honest zone other than where the validator belongs according to its actual location. For example, a compromised node in Germany could be assigned to the US zone.

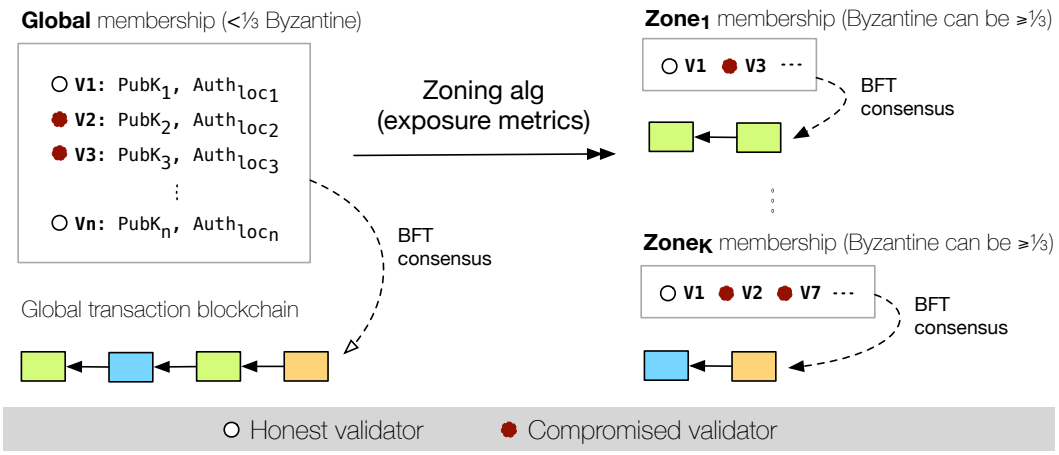


Figure 3.3: Nyle secure zoning. Nodes join the Global zone membership and claim a provable location. Based on these claims, anyone can obtain the local zones' memberships by running the zoning algorithm. The Global zone is secure, but local zones can be compromised.

This attack may increase zone exposure, especially in the case of leader-based consensus algorithms, because the leader might be outside the zone, breaking condition 5.

However, in the case of leaderless consensus algorithms, we argue that this is an attack that Nyle already accounts for. A minority of compromised nodes with a false location does not increase the exposure, because such a minority is indistinguishable from a minority of compromised nodes that are honest about their location, but arbitrarily delay their consensus messages. The latter attack is already comprised in our attacker model.

Securing zoning. To counter the two membership attacks we identified above, Nyle enhances the Global zone membership protocol as depicted in Fig. 3.3. There are two phases: Global zone membership registration and local zones membership computation. First, each validator can only register for the Global zone, and needs to declare at registration time its public key $\mathsf{Pub}_{\mathsf{K}_i}$ and an authenticated (signed) location claim $\mathsf{Auth}_{\mathsf{loc}_i}$. The specific location claim depends on the Lamport exposure metric. This section considers geographic location as the exposure metric; we defer the possibility of using other metrics to the discussion Section 3.9. Based on this information, the zoning algorithm places validators in local zones. Clearly, this ensures that all local zones' validators are a subset of the Global zone, ensuring condition 2.

The mechanism of making a location claim addresses the two membership attacks. Addressing attack 1, the mechanism prevents the attacker from amplifying its power in several zones, because each node, including compromised nodes, needs to commit to a single location. Addressing attack 2, each honest node can obtain an accurate location, by our assumption. Moreover, because each honest node independently declares and signs it with its private key, the attacker cannot craft the location of compromised nodes. Addressing attack 3 for geographical location claims, we use a system like VerLoc [74], which verifies the claimed

geographical locations using RTTs. Thus adversaries cannot spoof their location without detection.

For the second phase, we assume the Global zone blockchain stores a zoning policy given during the bootstrap phase. A zoning policy may specify, for example, to group nodes in zones by countries using the claimed coordinates, or to use autozoning with certain radiuses when the location claims contains RTTs between nodes (Section 2.6.3). When the Global registration ends, any node can traverse the global membership and apply the zoning policy as in Limix. Because the Global validator set is immutable and public, and the zoning algorithm is deterministic, each validator deterministically computes the same membership set per zone. The validators in each zone can then run consensus and maintain the zone's blockchain.

Nyle zoning vs sharding. Zoning might seem similar to the concept of Omniledger-style sharding, because both approaches work with a subset of the validator nodes for each zone, shard respectively. However, this is where the similarities stop. An important distinction is that Omniledger partitions the state between shards with no overlaps: there is one single shard that commits a given transaction. For this reason, it is crucial in Omniledger that no shard is insecure. In contrast to Omniledger sharding, in Nyle there are usually several zones that commit a given transaction to their blockchain. Local zones might be compromised, but the crucial point in Nyle is that the Global zone is always secure.

Epochs. The zoning protocol needs to accommodate zone changes. The zone membership can change when validators join or leave the system, or when validators fail indefinitely. Also, the Global zone might terminate compromised zones which provably misbehave, e.g., by signing an acceptance of double-spending. For these reasons, Nyle proceeds in epochs, just like related work [75, 76]. Nyle allows for membership changes to take effect at the beginning of the subsequent epoch.

3.4.2 Trust-but-verify transaction validation

The locality-based zoning above makes a major attack possible. At any time, an attacker might concentrate its efforts on compromising a zone. Indeed, the attacker may localize all of its validators in a single or a few zones, e.g., by physically deploying validators there, effectively controlling more than one-third of the validators in that zone. Globally, the attacker still controls fewer than one third of validators, but locally, he or she can surpass this threshold. As a cost for its exposure-limiting properties, Nyle must face this attack and counter it to protect transactions from remote failures, network partitions, and slowdowns.

To defend against this attack, Nyle designs a trust-but-verify architecture that offers the transaction sender and receiver a choice of exposures, one per zone. Individually, the sender and receiver each choose from which zone to trust transaction validations. The sender transmits its transaction for validation to all zones that overlap its location, including the sender's own

trusted zones. The receiver periodically polls its trusted zones to discover newly validated transactions. The Global zone is the only zone that must be trusted by all participants; however, participants may opt to trust other zones in order to reduce their exposure, proceed without waiting for other validations, and afterwards verify whether the Global zone validates these transactions as well.

Shortcomings: attacks and overheads. If we merely allow the sender to run the basic aforementioned protocol, we may enable two attacks. The first attack concerns overheads, whereby a compromised sender may choose to broadcast its transaction to all zones in an effort to overload them computationally by executing validations and storage-wise by storing blocks. A compromised receiver might poll any zone as frequently as it desires, in order to inflict computational and network overload. Nyle must limit the overhead that zone participants can impose.

The second attack concerns double spending. Double spending refers to the situation where the same UTXO is spent multiple times in transactions that are deemed valid. In Nyle, multiple zones validate the same transaction; thus, these zones contain records of the same output UTXO. A compromised sender may attempt to simultaneously spend its token in such zones, but in separate transactions, resulting in double-spending. Essentially, double spending breaks the blockchain strong consistency property. Zones need to be able to identify a potential double spending situation by determining the most recent state of the UTXO, and they need to be able to prevent double spending by disallowing concurrent state changes of that UTXO.

3.4.3 Protocol sketch

We now sketch a protocol that addresses the two attacks previously mentioned in Section 3.4.2, though it is not yet the full protocol. Fig. 3.4 depicts the process. We address the first attack by having each participant commit to a location. Each participant registers on the Global chain with its location and public key, just like validators do. The participant can also add to its registration block a list of trusted zones that cover its location. If this field is missing, all zones covering the participant's location are considered by default trusted by the participant. Possible location changes of the participant take place in the subsequent epoch. Only the zones covering the sender location (SL) are candidates to validate the sender's transactions, though the sender can choose to trust and send the transaction to just a subset of these zones. The Global zone always validates every transaction. The receiver can also poll only the zones covering the receiver location (RL).

If a KYC system is in place [40, 97], then it already enforces that RL and SL correspond to the real receiver and sender locations, potentially at a jurisdiction granularity. However, if this constraint is not present, the sender and receiver could in principle claim any location. The cost of them doing so is an increase in transaction exposure with the distance between their real location and the claimed location. Zones that fear high overheads due to a crowding herd

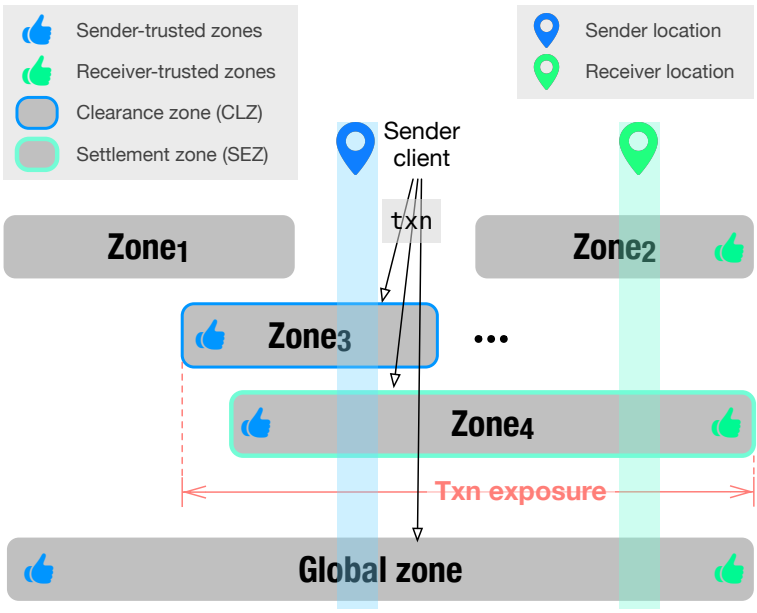


Figure 3.4: Nyle's trust-but-verify architecture. Nyle settles transactions in the smallest jointly trusted zone, called the SEZ. Participants can afterwards verify whether the Global zone also accepts the transaction. Nyle bounds overheads by requiring each participant to only impose loads on zones that overlap her location. Nyle avoids basic double spending with a CLZ per UTXO, that stores the definitive UTXO state and synchronizes all zones spending that UTXO. The transaction exposure is the combined exposure of the SEZ and CLZ.

of clients could increase transaction fees or implement any other rate-limiting mechanism.

The sender and receiver are ready to accept a transaction when a jointly trusted zone validates the transaction. Based on the previous paragraph, a jointly trusted zone "covers" both the sender's and receiver's locations SL and RL. There may be several such zones, for example in the figure Zone_4 and Zone_5 are trusted by both the sender and the receiver. Of these zones, we denote as the *settlement zone (SEZ)* the zone that validates the transaction first.

We address the second attack by having a designated zone per UTXO, called the clearance zone (CLZ), which is authorized to spend the UTXO. The CLZ should cover the SL, otherwise, per the protocol above, the sender cannot forward its transaction to the CLZ. All other zones validating the same transaction first synchronize with the CLZ and only afterwards run their independent validation of the transaction. The SEZ also needs to synchronize with the CLZ. Thus, the transaction's exposure is the combined exposure of the SEZ and the CLZ.

With this protocol sketch, we can already explain the basics of transactions. It would not be particularly convenient if the output UTXO implicitly had the same CLZ as the input UTXO, because the receiver may not trust the CLZ, yet would have his funds "locked" in that CLZ. It is natural, however, to use the SEZ for this purpose. Being a zone that is trusted by the receiver and that validates the transaction, the SEZ has the output UTXO, and becomes the output UTXO's CLZ. Moreover, anyone can compute the output UTXO's CLZ: it is smallest exposure

zone containing both RL and SL.

Although until now we described transactions with a single input UTXO and a single output UTXO, it is not too difficult to extend the model to multi-input and multi-output. A sender should first transfer all input UTXOs to the same CLZ (Section 3.6). Then, the sender prepares and broadcasts the transaction just like before. If all outputs are for the same receiver, the outputs' new CLZ is the SEZ. However, if there are outputs for different receivers R_i, each of them with its own set of trusted zones, we may naturally obtain several $\mathsf{SEZ_i}$, one per receiver. $\mathsf{SEZ_i}$ becomes the CLZ for output $\mathsf{UTXO_i}$, and corresponds to the smallest exposure zone common to the sender location SL and the receiver location $\mathsf{RL_i}$. Because all inputs have the same CLZ that approves the transaction at once, the transaction is atomic. The various SEZs do not interfere with atomicity: they are simply relevant for the spending of each of the output UTXOs.

There are, however, many challenges that this simple sketch protocol does not address. The SEZ and CLZ could become compromised at any point, and validate double spending, or become unresponsive. An unresponsive CLZ, in particular, might lock fund and censor the sender from spending their UTXOs. The next section dives more deeply into these challenges and the specifics of transactions. For simplicity, we present our protocols for single-input single-output transactions, and describe particular points for multi-input multi-output transactions when necessary.

3.5 Transaction confirmation protocol

This section describes the transaction protocol in Nyle, albeit with a simplification: For a transaction spending token $\mathsf{UTXO_x}$, we assume that the sender and all zones already know the CLZ of $\mathsf{UTXO_x}$. The next section Section 3.6 relaxes this assumption, by handling the choice of CLZ and CLZ transfers.

3.5.1 Challenges

Consistency. In a strongly consistent protocol, such as Nyle, it is essential that the nodes/-validators can identify the most recent state of an UTXO. Limix tracked the most recent state using a token per key. Nyle, however, differs from Limix in two fundamental ways. First, only one client can spend ("write") an UTXO, i.e., the UTXO's owner. Thus, there cannot be any conflicting spending, i.e., double-spending, at least not when the sender is honest. When the sender is compromised, however, it may attempt to double spend. Second, an UTXO can only be spent ("written") once, thus it has only three possible versions: non-existent, valid (unspent), and spent, which is the terminal state. The job of validators is to determine the definitive state of the UTXO.

This is where the CLZ becomes relevant, because the CLZ stores the definitive state of an

UTXO. Consider the transaction Charlie $\xrightarrow{\text{In:UTXO}_0,\text{Out:UTXO}_1,\text{CLZ:Zone}_1}$ Alice), with CLZ Zone_1. This means that Zone_1 stores by definition the latest state of the input UTXO_0. Nyle's CLZ protocol Section 3.6 ensures that the CLZ indeed holds the latest state of the input UTXO.

Compromised sender & zones. What makes it particularly challenging in Nyle to handle double spending are compromised zones. Typically, blockchains that rely on BFT ensure that shards are honest, which means shards guarantee the safety of consensus and disallow a compromised sender to double spend. However, this is not the case in Nyle. A compromised sender that colludes with a compromised (clearance) zone could succeed in double spending its UTXOs.

Handling compromised local zones is Nyle's price to pay for limiting transaction exposure. Small local zones immunize transactions from partitions, failures and slowdowns outside the zone, but they can be compromised by a powerful attacker. A compromised zone could break transaction safety, e.g., by allowing a malicious sender to double spend, by rewriting blockchain history, and liveness, e.g., by arbitrarily delaying replying to the receiver, the sender and other zones.

There are two ways Nyle tackles safety. First, Nyle uses a trust-but-verify architecture, where participants can always fall back on the safety of the Global zone. In principle this provision is sufficient, however, Nyle would not be very useful if compromised senders could always choose to double-spend by spending their funds through compromised zones. Thus, as a second provision, Nyle uses a protocol for establishing an UTXO's CLZ (Section 3.6), which the receiver has some control on, but jointly with the sender. The receiver can still perform a CLZ transfer to a compromised zone, however, this incurs additional costs and latency.

There are two ways in which Nyle tackles liveness. When we refer to the spending of an UTXO, the CLZ and the SEZ are the relevant zones for livenes. Nyle handles CLZ liveness issues through explicit CLZ revocation by the sender (Section 3.6), coupled with long liveness checkpoints by which the CLZ is expected to contact the Global zone. In case of no contact, the Global zone revokes the CLZ's tokens (Section 3.6). A compromised SEZ affects primarily the spending of output tokens, to which the same reasoning applies. The SEZ cannot prevent parties from learning about a transaction validation, because ultimately they learn from the Global zone, if not from one of the other common trusted zones of the sender and receiver.

3.5.2 The protocol

Sender. The sender in a transaction starts the process for getting the transaction confirmed, depicted in Fig. 3.5. Assume that the CLZ for UTXO_1 is Zone_3. To send UTXO_1 to the receiver B (Bob) (step 1 in Fig. 3.5), the sender A (Alice) prepares the transaction in Eq. (3.1), signs the transaction and sends it to a threshold of validators in $Zone_3$ via efficient broadcast, for example. The sender also appends its timestamp in the transaction, which protects the sender

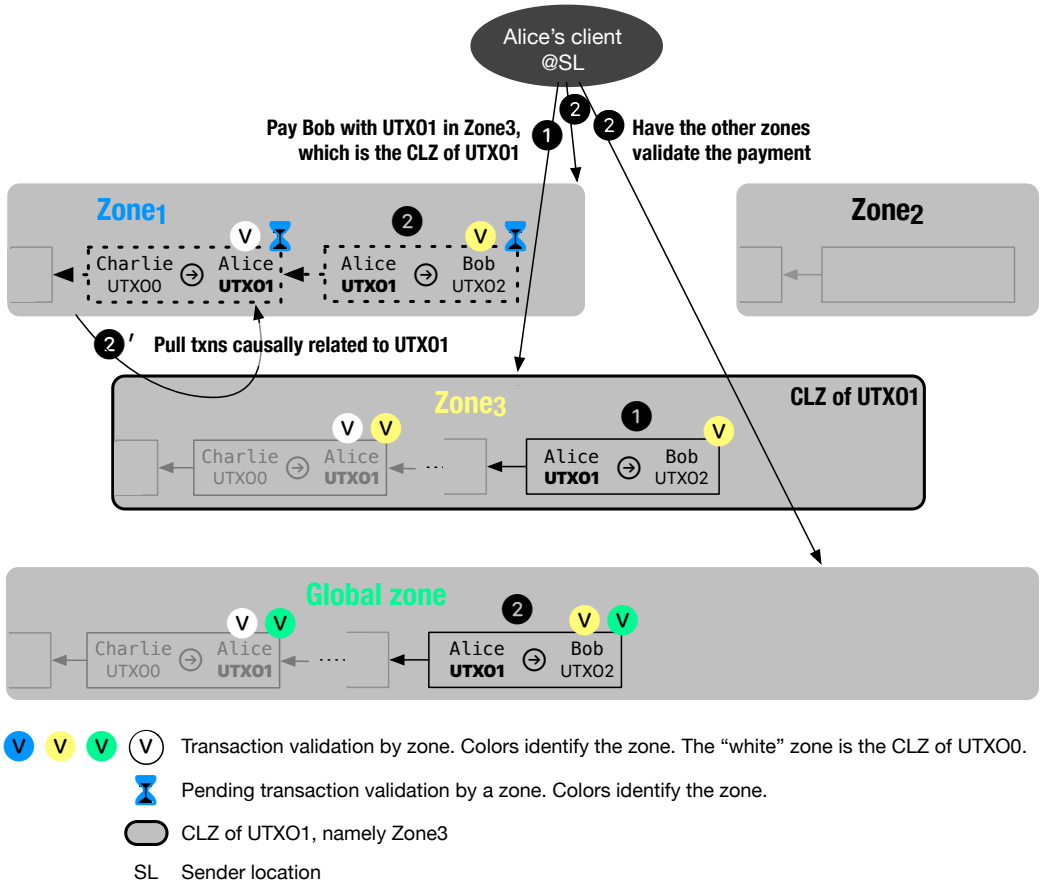


Figure 3.5: Transaction validation in Nyle. Alice sends UTXO_1 to Bob. (1) Alice submits the txn for validation to Zone_3, which is the CLZ of UTXO_1. (2) Alice submits the txn to other zones that overlap her location. The CLZ Zone_3 acts as a single point of synchronization for UTXO_1, assisting to prevent double-spending.

against framing attacks by malicious parties that replay the packet to incriminate the sender of double spending. By default, all zones that have the SL in their membership validate the transaction. However, only some of these zones may be trusted by both the sender and the receiver, according to their inputs in the beginning of the epoch, and then only those zones perform validation. The Global zone always validates all transactions.

$$\mathsf{Txn}_{A\to B} = \mathsf{Sig}_{\mathsf{PrivK}_A}(\mathsf{Sender}:A \,||\, \mathsf{Recv}:B \,||\, \mathsf{In}:\mathsf{UTXO}_1 \,||\, \mathsf{Out}:\mathsf{UTXO}_2 \,||\, \mathsf{Time}:\mathsf{ts}_A) \tag{3.1}$$

Zones. An UTXO's designated CLZ is the only zone that (1) Is guaranteed to have the latest state of the UTXO, (2) Is guaranteed to store all causally related transactions that lead to that UTXO's creation, and (3) Is authorized by the UTXO owner to modify the UTXO's state by spending it. Any other zone first needs to synchronize with the CLZ before validating a

transaction spending the UTXO. The CLZ is the first zone to run consensus on the transaction validity, and to do so it accesses the latest UTXO state, guaranteed to be on its blockchain. The other zones trusted by the receiver that validate the transaction initiate validation only after they synchronize with the CLZ and learn of the authenticated validation of the CLZ.

The CLZ may be compromised, however, like any zone that is not global. This means not only that the CLZ may approve double spending, but also that the CLZ may not be live. It would be undesirable that larger zones or especially the Global zone to be held up forever waiting to synchronize with the CLZ in order to do anything. Nyle handles CLZ liveness issues by installing a long timeout - on the order of several hours or days - by which the CLZ is expected to respond and make progress, and if it fails to do so, the Global zone can revoke the token ownership of the CLZ and recover the tokens in the Global zone (Section 3.6). If the CLZ only censors the sender, the sender can explicitly revoke the token ownership from the CLZ, which is verified and carried out by the safe Global zone (Section 3.6).

As a transaction example, Fig. 3.5 depicts Zone_3 being the CLZ for UTXO_1. The CLZ Zone_3 validators run consensus on the transaction validity. First, each validator checks that Zone_3 is the rightful CLZ for UTXO_1 by verifying the transaction that output UTXO_1, i.e., Charlie's and Alice's locations, and their common trusted zone with the smallest exposure. The CLZ also checks that it stores all valid causally related transactions outputting UTXO_1. Next, the validator checks that UTXO_1 was not already spent in the CLZ, i.e., Zone_3's blockchain does not contain an earlier transaction with UTXO_1 as input. Finally, the validator checks that UTXO_1 belongs to Alice and Alice's signature is valid. Then the validator appends its decision Accept/Reject to the transaction validation and signs it.

$$\mathsf{Validation}_{\mathsf{V}} = \mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{V}}}(\mathsf{Txn}_{\mathsf{A}\rightarrow\mathsf{B}}||\mathsf{Accept}) \tag{3.2}$$

When a threshold of validators sign the same decision for a transaction, all validators append it to Zone_3's blockchain, in other words they publish the transaction and the validation.

$$\mathsf{Validation}_{\mathsf{Zone}_3} = \mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{Zone}_3}}(\mathsf{Txn}_{\mathsf{A}\rightarrow\mathsf{B}}||\mathsf{Accept}) \tag{3.3}$$

Once the CLZ publishes a validation, Alice can proceed to get her transaction validated by all zones that it trusts and overlap the SL. In this example, these are Zone_1 and the Global zone, mentioned in the transaction Eq. (3.1). Specifically, both Alice and the CLZ forward the CLZ validation to some validators in these other zones. Note that transaction forwarding by both Alice and the CLZ is needed in case one of these parties is compromised and decides to stall the protocol. Duplicate transactions are not an issue, because anyone can recognize and ignore them. We discuss in detail in Section 3.5.3.

The steps below apply for each of the sender trusted zones: Zone_1 and the Global zone. Each of the validators in a sender trusted zones first verify that the validation signature $\mathsf{Validation}_{\mathsf{Zone}_3}$ is correct. Then, if the zone's blockchain does not yet have a record of UTXO_1, the zone

contacts the CLZ to fetch those transactions that causally determine the issuance of UTXO_1. Then, the zone independently validates all these transactions on its blockchain. Finally, each validator in such a zone runs the verification protocol above for $\mathsf{Txn}_{\mathsf{A}\to\mathsf{B}}$ and adds the validation to the zone blockchain, together with the authenticated CLZ validation. The CLZ validation is necessary to show that the CLZ knows about the transaction and validated it, regardless of the decision. Note that the zone's decision could be different from the CLZ's decision, for example if the CLZ is compromised. The transaction validation that both the sender and receiver should follow is that of the SEZ, which by definition is guaranteed to be one of the zones trusted by the sender. In case any party, including the sender or receiver, believes the SEZ's decision included in the transaction validation is wrong, then the party can report the transaction and wait for the Global zone validation.

$$\mathsf{Validation}_{\mathsf{Zone}_\mathsf{X}} = \mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{Zone}_\mathsf{X}}}(\mathsf{Txn}_{\mathsf{A}\to\mathsf{B}}||\mathsf{Accept}||\mathsf{Validation}_{\mathsf{Zone}_1}) \tag{3.4}$$

In the example in Fig. 3.5, Alice sends the Zone_3 yellow validation to Zone_1 and Global zone. The Global zone can proceed right away to validate the transaction, because it has UTXO_A on its chain. However, Zone_1 needs to first fetch $\mathsf{Txn}_{\mathsf{C}\to\mathsf{A}}$ from the CLZ Zone_3 and validate it, before validating $\mathsf{Txn}_{\mathsf{A}\to\mathsf{B}}$.

Periodic synchronization. Periodically, bigger zones pull transactions from smaller overlapping zones to ensure their states are not too out-of-sync. Generally speaking, smaller zones can have a more up-to-date state because their validators communicate faster, thus finalize new transactions faster. Because validators broadcast transactions, bigger zones should eventually receive these transactions and confirm them. However, attacks that we discuss below or disconnects might stall broadcast. For this reason, validators in any zone periodically check the zone's blockchain against smaller overlapping zones' blockchains, pull and run the validation protocol on new transactions.

There are some situations (see Section 3.5.3 and Section 3.6.6) when the Global zone detects that another zone is compromised and decides to suspend it. The Global zone immediately pushes such zone or membership changes to all other zones, so that they neither contact nor accept transactions from the compromised zones anymore.

Receiver. The receiver contacts some zones it trusts to learn about new validations. The receiver can keep secret what zones it trusts for validations, because only the receiver waits for validations from these zones. The receiver's choice determines the other part of the transaction's Lamport exposure, depicted in Fig. 3.4.

Even though the Global zone is the only zone that is guaranteed to be trustworthy, the receiver is not required to always wait for the always-safe global transaction validation. For instance, the recipient may determine that a local, small zone validation is trustworthy enough for

#	Participant honest(✓) / compromised(×) / unknown(∗)				Nyle property guarantee yes(✓) / no(×) / N/A [reason]						
					Safety			Liveness*			Exposure
	Sender	Recv	CLZ	SEZ	Sender	Recv.	Zones	Sender	Recv.	Zones	
1.	✓	*	✓	✓	✓	✓	✓	✓	✓	✓	SEZ
2.				×		✓[S2]			✓[L1]		Global
3.	✓	*	×	✓	✓[S1]	✓	✓	✓[L2]	✓	✓[L2]	SEZ
4.				×		✓[S2]			✓[L1,L2]		Global
5.	×	*	✓	✓	N/A	✓	✓	N/A	✓	✓	SEZ
6.				×		✓[S4]			✓[L1]		Global
7.	×	*	×	✓	N/A	✓[S3]	✓[S3]	N/A	✓	✓[L1,L3]	SEZ
8.				×		✓[S3]			✓[L1]		Global

Table 3.1: Nyle security properties. On the left side there are the parties involved in any transaction: the sender, the receiver, the CLZ and the SEZ. On the right side of the table we show the guarantees for a honest sender, honest receiver, and for any honest zones involved in the transaction. Nyle offers guarantees only for uncompromised parties, otherwise the guarantee is N/A. For more complex cases, we provide a brief reasoning, referenced by reason ID in Table 3.2.

minor payments or transactions between trusted parties. Eventually, because of regular synchronization, the Global zone verifies all transactions on its blockchain, making them accessible to the world. Because zones may be malevolent, the receiver forwards transactions meant for it to the Global zone so that it can later determine whether the Global zone accepts them. Duplicate transactions are not a problem since they are identifiable and easily ignored. Receivers may elect to continue waiting after getting validations from smaller zones and afterwards determine that the Global zone has also confirmed the transactions.

The receiver checks all received validations $\mathsf{Validation}_{\mathsf{Zone}_i}$. First, it checks all the signatures, to defend against packets crafted or tampered with by an attacker that attempts to spend in the sender's name or attempts to validate transactions impersonating a zone. Also, the packet could be incorrectly signed by the SEZ, in an attempt to deprive the receiver of liveness. Next, the receiver verifies that the zone sending the validation, i.e., the SEZ, and the CLZ, both accepted the transaction. When either the signatures are incorrect, or the validations of the SEZ and CLZ do not match, then one or more parties among the sender, the CLZ, the SEZ are compromised, and the receiver follows the steps in Section 3.5.3 to decide which validation to trust, and whether it should inform the Global zone of the behavior. In brief, the receiver can always verify any validation with the Global zone, and can ultimately rely on the Global zone for liveness.

3.5.3 Security argument

We analyze the guarantees Nyle provides to the participants during attacks, focusing on safety, liveness, and exposure. Recall that any of the system participants can be compromised: sender, receiver, and zones, by having more than 1/3 of validators compromised. Thus, our analysis

Reason ID	Description
S1	The sender can forward an already validated transaction to cancel CLZ history rewrite.
S2	The transaction should be accepted, the receiver detects the bogus reject of a compromised SEZ.
S3	Zones and receiver notified of double spending by the Global zone.
S4	The transaction must not be accepted, the receiver detects the bogus accept of a compromised SEZ.
L1	The Global zone eventually replies with an authentic validation.
L2	The sender replaces a censoring CLZ via a CLZ transfer (Section 3.6).
L3	The zones contact the Global zone to intervene if the CLZ is unresponsive.

Table 3.2: Brief explanation for the reasons behind some security guarantees. Referenced from Table 3.1.

accounts for the fact that each participant can only rely on the actions it executed itself and on the actions of the always secure Global zone to achieve the desired properties. Nyle provides guarantees only to those participants that follow the protocol, such as honest participants or (temporarily) well-behaving compromised participants.

Throughout the analysis we assume that all honest parties perform signature checks, which defends against signatures forged and packets crafted or modified by an attacker. Thus, we do not discuss these attacks further.

Table 3.1 summarizes the results of our security analysis. On the left side of the table we enumerate the parties involved in any transaction: the sender, the receiver, the CLZ and the SEZ. The receiver appears in the table as a party. However, whether the receiver is honest or not is relevant solely for the receiver's guarantee, but does not influence anyone else's guarantees. Thus, we do not differentiate between honest and compromised receivers. The SEZ is by definition the zone trusted by both the sender and the receiver and which replies first. On the right side of the table we show the guarantees for the sender, the receiver, and for any honest zones involved in the transaction. We analyze the properties by case number.

At the end of this section, we also review framing attacks, where compromised parties attempt to make other parties look compromised.

Honest sender. All the cases presented here have an honest sender that does not double spend. Case #1 in Table 3.1 presents an honest sender, CLZ and SEZ. in this case, Nyle ensures safety, liveness under weak synchrony, and a transaction Lamport exposure set by the SEZ. Analyzing safety, for the honest zones the underlying BFT consensus protocol guarantees safety, in particular also in the CLZ and SEZ zones, which are honest by case #1's assumptions. Any zone pulling transactions from another zone is guaranteed to pull safe transactions. The sender is guaranteed safety, because, by case #1's assumptions, neither the CLZ, nor the SEZ, misbehave by deleting blockchain history, etc. Finally, in conjunction with an honest sender that does not double spend, Nyle guarantees safety to the receiver.

Analyzing liveness, we only need to consider what happens when participants are faulty/unre-

sponsive, because by our assumption they are already not compromised. When the sender is unresponsive and does not send any transaction, then there is no property to guarantee. However, if the sender looses liveness after sending its transaction, we need to consider which parties learn about the transaction. Under a weakly synchronous network model, the CLZ eventually learns about the sender's transaction. Furthermore, with this communication model, the CLZ is guaranteed to be live and pushes the sender's transaction to any zone requesting it, guaranteeing liveness to other zones. Finally, because the SEZ is live, the receiver eventually receives a transaction validation, thereby ensuring liveness for the receiver.

Analyzing exposure, recall that the SEZ is by definition the first zone that validates the transaction trusted by both the sender and the receiver. Because the SEZ provides an answer eventually and, by case #1's assumption, it is not compromised, it trivially bounds the transaction Lamport exposure.

Case #2 considers an honest sender and CLZ, but a compromised SEZ. The compromised SEZ affects only the receiver's guarantees, because no other party depends on replies from the SEZ. Regarding safety, the transaction should be accepted because it is not a double spend, however, the compromised SEZ that replies first might wrongly reject the transaction (reason S2 in Table 3.2). Fortunately, the receiver can detect the bogus reject, because the SEZ cannot craft a valid proof of double spend in the sender's name. Regarding liveness, it might seem that the receiver is guaranteed liveness because the SEZ replies by definition. However, the SEZ might send a bogus validation, e.g., with an invalid signature, which the receiver ignores. Nevertheless, the sender and the CLZ forward the transaction to other zones and, eventually, some zone or in the worst case the Global zone replies with an authentic validation - reason L1 in Table 3.2. This situation increases the Lamport exposure of the transaction to the first SEZ that sends a correct reply.

Next, in case #3, we consider an honest sender and SEZ, but malicious CLZ. The CLZ interacts with the sender and with other zones. Regarding safety, the CLZ could validate the sender's transaction but rewrite the blockchain history at any point, removing the sender's transaction. However, the sender can simply store its validated transaction locally and send it to other zones - reason S1 in Table 3.2. Regarding liveness, the CLZ could refuse to reply or send invalid replies to the sender and other zones, in which case the sender can transfer its CLZ (reason L2 in Table 3.2), potentially requiring an intervention via the Global zone. We address CLZ transfer in detail in the next section. In the worst case, the CLZ transfer might increase the transaction exposure to Global exposure.

When the sender is honest but both the CLZ and the SEZ are compromised, as in case #4, we need to additionally evaluate the receiver guarantees. The receiver safety argument is the same as in case #2 above. Regarding liveness, the reasoning combines the cases #2 and #3 above, and relies on a potential CLZ transfer initiated by the sender (reason L1), after which the receiver eventually obtains a validation from the Global zone (reason L2).

Compromised sender. We now consider a compromised sender that may attempt to double spend. Case #5 presents a compromised sender, but honest CLZ and SEZ. Safety is guaranteed for all honest parties because the CLZ stops any double spending attempt, and neither CLZ nor SEZ rewrite history. The malicious sender has no influence on liveness, so liveness is also guaranteed.

Next, in case #6, the SEZ becomes compromised along the sender. As we have seen in cases #2 and #4, a compromised SEZ affects the receiver guarantees. The compromised SEZ might accept the double spending transaction in an attempt to break safety, however, the receiver can find the CLZ in the transaction, observe that the CLZ rejected the transaction, and can check the CLZ's reasons, which reveal double spending transactions signed by the sender (reason S4 in Table 3.2). The liveness and exposure reasoning are similar to case #2: the CLZ is honest and forwards the transaction to other zones and, in the worst case, the Global zone provides an authentic validation. In the worst case, the exposure is Global.

The final two cases, #7 and #8, both consider a compromised sender and CLZ that collude with each other. If the sender follows the protocol, the sender obtains the same guarantees as in case #4. However, the sender might double-spend and the CLZ might validate the double spending. When zones ask the CLZ for transactions, the CLZ might equivocate and present them with different blockchain forks, each containing just one of the sender's transactions, without any trace of double spending. Thus, the CLZ can trick some honest zones, such as the SEZ, to accept and validate one of the sender double-spending transactions, whereas other honest zones accept and validate the other double-spending transaction. The Global zone eventually detects the double-spending (reason S3 in Table 3.2), thanks to the periodic synchronization and the trust-but-verify principle. In case of double-spending, the Global zone can issue a refund. Moreover, the Global zone could take further corrective actions against the malicious validators that approved the double-spending. Describing these actions is outside the scope of this thesis. The malicious user could also suffer punitive measures, e.g., by needing to increase its collateral or being banned from the system. Refund and punitive measures are, however, outside the scope of this thesis.

The compromised CLZ also affects liveness. First, neither the sender nor the CLZ might forward the transaction to other zones. However, this is not a liveness attack, because no other party (other than the Global zone) actively waits on learning about this transaction. During periodic synchronization, if the CLZ does not reply to the Global zone within Δ, the Global zone could temporarily disable the CLZ for suspected compromise. The CLZ might forward the transaction but might refuse to send or send rogue causally related transaction validations, which prevents honest zones from validating the transaction. In this case, the honest zones might request the Global zone intervention and retrieve the necessary transactions via the Global zone. Due to this, the transaction exposure is, again, Global.

Finally, in case #8, the sender, CLZ and SEZ are all compromised. The SEZ affects just the receiver guarantees. Just as in case #7, the receiver is guaranteed safety only by the Global

zone (reason S3), but not because the SEZ was tricked, but because the SEZ is compromised. For liveness, just as in cases #2 and #6, the receiver eventually received an authentic reply from the Global zone (reason L1). The exposure is Global.

Framing attacks. Framing attacks refer to actions that compromise parties take in order to make other parties look compromised. We first remark that nobody can impersonate the sender, receiver, zone or validator, because they cannot fake signatures. Thus, the only tools available are packet insert, drop, delay. An attacker could replay (packet insert) the sender transaction in order to frame the sender of double spending. However, the sender adds a timestamp in the transaction, so a replay is trivially detected as a duplicate. An attacker could frame a zone by replying (e.g., to a receiver) faster than the zone, likely with an incorrectly signed validation. However, the recipient simply ignores such incorrect validations. Finally, no action initiated by the receiver is critical to the protocol, thus the protocol does not react on receiver inactivity. Hence, framing the receiver is not possible.

3.6 CLZ Choice and Transfer Protocol

The section above rests on the assumption that there exists a single CLZ per UTXO, and this CLZ is known to everyone. This section describes how Nyle chooses an UTXO's CLZ to meet the CLZ properties. Next it, describes the protocol for a CLZ transfer, which happens when a sender wishes to spend its token in a different zone than its default CLZ. The section also describes Nyle's strategy for handling CLZs and zones that lose liveness. Finally, it analyzes the security of the CLZ protocol.

3.6.1 CLZ properties

It is crucial that there exists a publicly known unique CLZ per UTXO. The CLZ of an UTXO stores the latest state of the UTXO and acts as a synchronizer for other zones that need to pull the latest state relevant for that UTXO. Nyle needs to ensure that an attacker cannot equivocate about the CLZ of an UTXO or create the situation where different honest parties believe an UTXO has different CLZs. If the attacker managed to create such a situation, the attacker could trivially double-spend via honest CLZs.

The default CLZ of an UTXO should be a zone trusted by the owner of the UTXO. The rationale is that the owner would most likely prefer to store and clear its UTXOs via CLZs it trusts. Otherwise, the owner can change the CLZ, however, the change incurs a Global exposure and further overheads, as we explain later in the section.

Finally, the CLZ should cover the owner's location. Otherwise, as per the previous section describing overheads, the owner would not be able to submit transactions for validation to the CLZ, and again would need to first perform an expensive CLZ transfer. This causes a potentially unnecessary increase for the exposure of the transaction spending the UTXO.

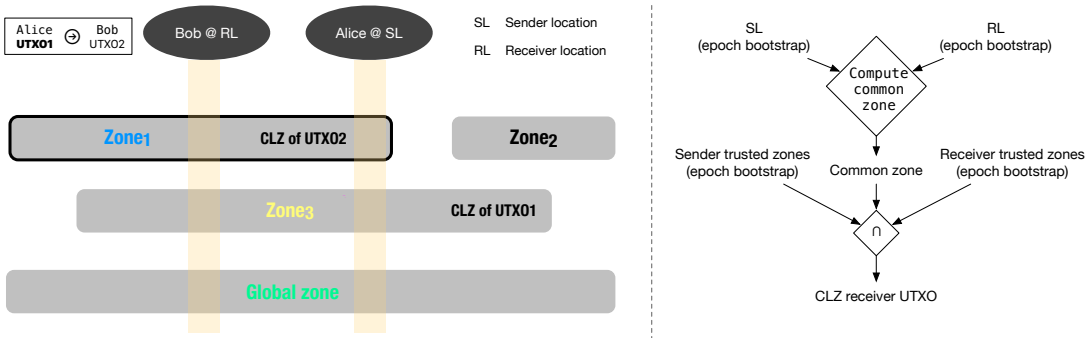


Figure 3.6: Nyle's choice of CLZ. The output UTXO_2 is assigned as CLZ Zone_1, which is the common trusted zone of Alice and Bob with the smallest exposure. The CLZ is uniquely determined by participant values provided in the beginning of each epoch.

3.6.2 Choosing the per-UTXO CLZ

Given the requirements above, Nyle designates the CLZ as follows. Assume that all participants know the CLZ of a transaction's input UTXO; at bootstrap all UTXOs have a known CLZ, which is the Global zone, for example. The CLZ that Nyle assigns to the output UTXO is the zone with the smallest exposure trusted by both the sender and the receiver. This CLZ might be the same as the SEZ, but not always: It could be that the SEZ, which is the commonly trusted zone that replies quickest, has a bigger exposure than the CLZ.

Fig. 3.6 the same transaction example as earlier, where Alice's transaction to Bob takes as input $UTXO_1$, outputs UTXO_2, and uses Zone_3 as CLZ, which is the known CLZ for UTXO_1. Then the output UTXO_2 is assigned as CLZ Zone_1, which is the common trusted zone of Alice and Bob with the smallest exposure.

We now argue that this CLZ choice fulfills the CLZ properties above. First, anyone can compute the CLZ by knowing the zone placement and membership, sender and receiver location, which are published on the Global blockchain at the beginning of the epoch. Second, the CLZ is chosen as a zone trusted by the receiver, which is the UTXO owner. Finally, all zones trusted by the receiver need to cover the RL, so by construction the CLZ covers the UXO owner's location.

3.6.3 CLZ transfer protocol

The CLZ transfer protocol is useful when the sender changes location and wants to spend at the new location its UTXOs created in other locations. Or it could happen when a zone censors a sender from spending its UTXOs, thus the sender changes the clearance zone.

The CLZ transfer is in fact a special type of transaction. Intuitively, the UTXO is spent at the old CLZ, and replaced by a new UTXO at the new CLZ. The advantages of representing CLZ transfers as transactions are uniformity, and trivially answering preliminary questions about incentives, i.e., client can easily "pay" for CLZ transfers.

In this case, the sender first needs to follow the protocol for a CLZ transfer. Once the CLZ transfer completes, all parties in question validate the transaction following the same protocol as for an intra-zone transactions (Section 3.5).

Guaranteeing an unique CLZ per UTXO. The most challenging aspect of a CLZ transfer is to preserve the safety property that only one CLZ is authorized to spend a certain UTXO. Imagine a compromised sender colluding with a compromised CLZ to transfer the CLZ to not just one, but many other honest zones. Then each of these honest zones believes it is the rightful CLZ, leading to honest zones authorizing transactions for the same UTXO, effectively authorizing double spending. Even though the Global zone eventually detects the fraudulent CLZ transfer, the attack breaks Nyle's safety guarantees: When the CLZ is honest, according to Section 3.5.3 and cases #1-2 and #5-6 in Table 3.1, then safety violations are immediately detected (reasons S2 and S4 in Table 3.2), without needing to contact the Global zone.

The way Nyle prevents this attack is to have all CLZ transfers proceed via the Global zone. The disadvantage is that the exposure of CLZ transfers is Global. By having all zone transfers proceed via the Global zone, a colluding compromised sender and CLZ cannot trick honest zones in believing they all are the new CLZ.

Consider the same transaction as before, Alice spending UTXO_1 to Bob, who receives UTXO_2 (Eq. (3.1)), with Zone_1 being the CLZ of UTXO_2. Assume that Bob changes his location and wants to perform a CLZ transfer for UTXO_2 from Zone_1 to Zone_2. Zone_2 can compute the CLZ for UTXO_2 like any other party, by using the sender and receiver location, and the zoning computed during the epoch bootstrap.

Bob executes the CLZ transfer. First, Bob asks the current CLZ Zone_1 to validate the CLZ transfer (Eq. (3.5)) - step 1 in Fig. 3.7, and provides this validation to the Global zone. Then, the Global zone checks whether to accept the CLZ transfer by running consensus on the authenticity of the Zone_1 CLZ transfer validation, then checks via consensus whether it all causally related transactions outputting UTXO_2, which it has (otherwise the zone needs to pull and validate them, like before), and stores the CLZ transfer validation on its chain (step 2 in Fig. 3.7 and Eq. (3.6)). Finally, Bob sends the Global zone validation to Zone_2, which performs the same checks as the Global zone (step 3 in Fig. 3.7). Zone_2 misses some causally related transactions, which it pulls from the Global zone in step 3'. Zone_2 stores on the chain its own validation Eq. (3.7).

When does the CLZ transfer take effect? Another critical point is ensuring there is a well-defined moment when the transfer takes effect. If the former CLZ Zone_1 is malicious, it might block other zones in finding out about the CLZ transfer. Or it might ignore that the CLZ transfer happened and rewrite history. If some parties might observe the transfer sooner than others, a malicious sender can exploit this situation and double spend in "all legitimate" CLZs.

For this reason, CLZ transfers take effect once the Global zone validates Zone_1's commitment

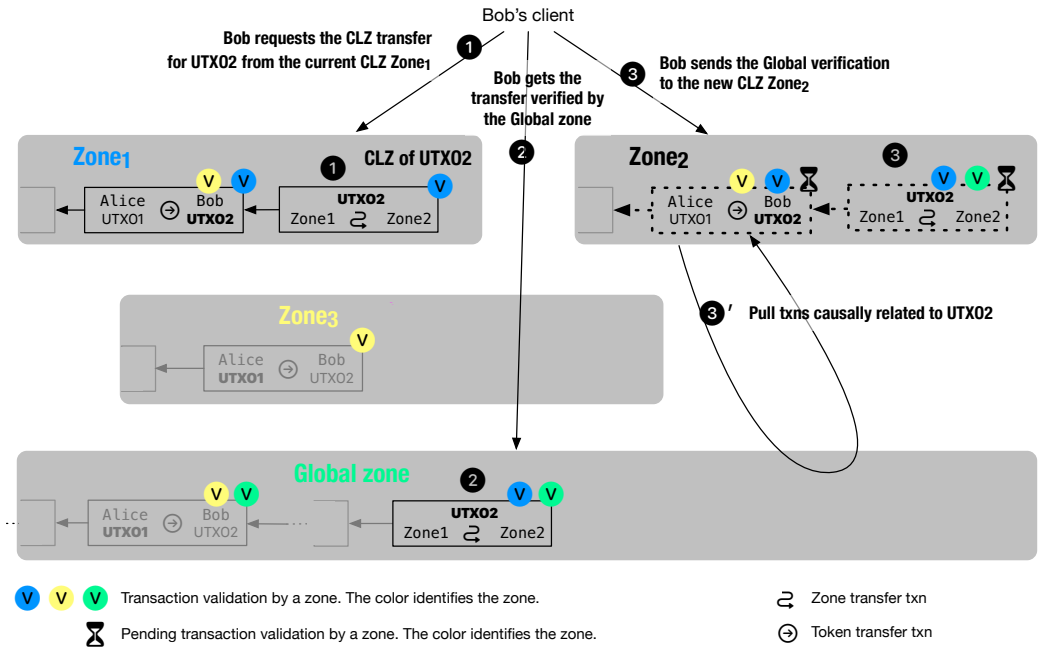


Figure 3.7: Bob transfers the CLZ for UTXO_2 from Zone_1 to Zone_2. For safety, all CLZ transfers proceed via the Global zone.

to relinquish the UTXO and the Global zone commits a record stating that it is now the owner of the UTXO. Because the global zone is definitive, no party can challenge the CLZ transfer. Even if Zone_1 is compromised and tries to double-spend the UTXO in two different transfers, given that the double-spend has to go through the Global zone, the Global zone detects it and ignores any second-spending attempts by Zone_1.

$$\begin{aligned}\mathsf{ReqTransferCLZ}_{\mathsf{Zone}_1\to\mathsf{Zone}_2} = &\mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{Zone}_1}}(\mathsf{Sig}_{\mathsf{PrivK}_B}(\mathsf{UTXO}:\mathsf{UTXO}_2 \,||\\ &\mathsf{Old}_{\mathsf{CLZ}}:\mathsf{Zone}_1 \,||\, \mathsf{New}_{\mathsf{CLZ}}:\mathsf{Zone}_2 \,||\, \mathsf{Time}:\mathsf{ts}_B))\end{aligned} \tag{3.5}$$

$$\mathsf{GReqTransferCLZ}_{\mathsf{Zone}_1\to\mathsf{Zone}_2} = \mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{Global}}}(\mathsf{ReqTransferCLZ}_{\mathsf{Zone}_1\to\mathsf{Zone}_2}) \tag{3.6}$$

$$\mathsf{TransferCLZ}_{\mathsf{Zone}_1\to\mathsf{Zone}_2} = \mathsf{Sig}_{\mathsf{PrivK}_{\mathsf{Zone}_2}}(\mathsf{GReqTransferCLZ}_{\mathsf{Zone}_1\to\mathsf{Zone}_2}) \tag{3.7}$$

3.6.4 Multi-input multi-output transactions

CLZ transfers are essential for transactions that spend multiple inputs. Multi-input multi-output transactions proceed in two steps. First, unless the inputs have the same CLZ, the sender(s) transfer all inputs to the same CLZ, via separate CLZ transfer transactions. These

transfers are not guaranteed to take place simultaneously, in the sense that some may succeed and some may experience liveness issues, e.g., due to the CLZ censoring a sender and may take much longer to complete (Section 3.6.5). Second, the sender(s) spend the inputs in the same transaction by producing the required signatures. The outputs can independently be assigned to different CLZs, depending on the sender and receiver locations, as explained in Section 3.4.3.

All inputs in the transaction share the same fate. If only one of the inputs is a double-spend, then honest zones reject the transaction, regardless of whether some inputs are not double-spends. However, it could happen that unwise clients that trust compromised zones accept the transaction, and attempt to use their output UTXO. In fact, the case of unwise clients is no different than a single UTXO transaction where the UTXO is a double-spend, and clients mistakenly trust the transaction. Nonetheless, the Global zone rejects their attempt, as it does with the multi-input transaction.

3.6.5 Censorship and liveness attacks by the CLZ

One of the reasons behind CLZ transfer is that the CLZ censors the sender by refusing to clear her transactions. However, such a compromised CLZ Zone_2 might also refuse to let Bob change the CLZ, which means that Bob may be unable to provide a CLZ transfer validation (Eq. (3.6)). In this case, Bob should simply provide the Global zone with the authenticated request for CLZ transfer signed by Bob (Eq. (3.5)). The Global zone then contacts Zone_1 to validate and commit a record of CLZ transfer, as requested by the sender. The Global zone then makes the same checks from the previous paragraph and validates the clearance zone transfer.

There is the possibility, however, that Zone_1 is unresponsive even to the Global zone's request of CLZ transfer. Of course, the zone might simply be just slow, or might be compromised and disrupt liveness. Nyle handles such liveness issues by installing a long timeout - on the order of several hours or days - by which the CLZ is expected to respond and make progress, and if it fails to do so, the Global zone can revoke the token ownership of the CLZ and recover the tokens in the Global zone.

An analogy would be to consider that UTXOs always "live" in the Global zone by default in the long term, but the global zone can lease them for limited times to the smaller zones. Smaller zones can renew their leases on the UTXOs they control (by being the CLZ) by making progress and checking in with the Global zone regularly, but if a smaller zone loses liveness and stops checking in, the lease eventually expires after a long timeout, and the Global zone effectively revokes control of the small zone's UTXOs back to itself.

#	Actor honest(✓) / compromised(×)			CLZ transfer guarantee [reason] yes(✓) / no(×) / non-applicable(N/A)							
				Safety				Liveness*			
	Sender	Old CLZ	New CLZ	Sender	Old CLZ	New CLZ	Honest Actor	Sender	Old CLZ	New CLZ	Honest Actor
1.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2.			×	✓[S1]		N/A	✓[S1]	✓[L1]		N/A	✓[L2]
3.	✓	×	✓	✓[S2]	N/A	✓	✓[S2]	✓[L1]	N/A	✓	✓[L2]
4.			×	✓[S1,S2]		N/A	✓[S1,S2]			N/A	
5.	×	✓	✓	N/A	✓	✓	✓	N/A	✓	✓[L2]	✓[L2]
6.			×			N/A	✓[S1]			N/A	✓[L2]
7.	×	×	✓	N/A	N/A	✓[S3]	✓[S3]	N/A	N/A	✓[L2]	✓[L2]
8.			×			N/A	✓[S1-S4]			N/A	✓[L2]

Table 3.3: Nyle security properties. The left side enumerates are the parties involved in CLZ transfers: the sender, the old CLZ, and the new CLZ. The right side summarizes the guarantees for the sender, the old CLZ, the new CLZ, and for any other honest actor (client or zone). More complex cases provide a brief reasoning for the guarantee, referenced by reason ID in Table 3.2.

Reason ID	Description
S1	The actor uses the Global zone validation to invalidate CLZ history rewrite.
S2	False claim of not being CLZ discovered if zone cannot show valid contrary claims.
S3	Double CLZ transfer prevented by the Global zone.
S4	Actor detects bogus CLZ transfer because transfer has invalid global signature.
L1	The sender eventually transfers CLZ away from the compromised CLZ via Global.
L2	Periodically, all zones and clients refresh their CLZ registry via Global.

Table 3.4: Brief explanation for the reasons behind some security guarantees. Referenced from Table 3.3.

3.6.6 Security argument

We now analyze the security of the CLZ transfer protocol. Because the CLZ transfer takes place between the sender, the old and new CLZ, we analyze what happens when these parties are compromised. Specifically, we look at the safety and liveness guarantees for these parties, and also for any other honest actor (participant or zone). Once the CLZ transfer completes and is safe, the security guarantees from Section 3.5.3 apply for the transaction validation.

Table 3.3 summarizes the guarantees. The left side of the table presents the parties involved in the CLZ transfer: the sender, the old and new CLZ, and considers cases where they are compromised. The right side of the table presents the safety and liveness for each of these parties, and for any other honest actor (participant or zone). Just as before, some guarantees have brief explanations, referenced by the reason ID in 3.4.

An attacker can attempt a number of safety and liveness attacks. A compromised client could attempt to transfer the CLZ to multiple zones, breaking the property that each UTXO has a single CLZ at any moment. Compromised zones can also rewrite history, falsely pretend to be

or not be the CLZ, allow double CLZ transfers. Compromised clients cannot affect liveness, but zones can by refusing to process a client's request of zone transfer, or refusing to serve another zone's request for missing transactions. Compromised parties can always collude with the purpose to amplify their attack, or attempt to frame an honest party of misbehavior.

We present our analysis looking at all combinations of compromised parties, by case number in Table 3.3. Our analysis accounts for the fact that each participant can only rely on the actions it executed itself and on the actions of the always secure Global zone to achieve the desired properties. Nyle provides guarantees only to those participants that follow the protocol, such as honest participants or (temporarily) well-behaving compromised participants. Throughout the analysis we assume that all honest parties perform signature checks, which defends against signatures forged and packets crafted or modified by an attacker. Thus, we do not discuss these attacks further.

We split the cases in two categories: those with an honest sender and those with a compromised sender. As a general remark, when the sender is honest, the guarantees for the sender, and the reasoning behind them, are the same as the guarantees for any other honest actor (column 5 and 8, 9 and 12 in Table 3.3). Also, the two CLZs never communicate directly, but via the safe Global zone. thus, the two CLZs cannot inflict damage on each other.

Honest sender. An honest sender does not attempt to perform a double transfer of an UTXO's CLZ. Because only the UTXO owner can legitimately transfer the CLZ, no other party can initiate a CLZ transfer without being detected via signature check.

Case 1 represents the case when all parties are honest. Trivially, there is no attack in this case, and safety and liveness holds for all honest participants.

In case 2 there is only one compromised party, namely the new CLZ. A compromised CLZ could accept the new role and add the CLZ transfer on its ledger, but omit to pull all causally related transactions that trace the UTXO's provenance. This, however, only hurts the zone itself, because, in order to validate transactions with this UTXO, a well-behaved zone needs to pull causally related transactions.

The new CLZ could accept the CLZ transfer, but later rewrite history and pretend it is not the CLZ. This affects the sender that attempts to spend its funds in the new ClZ, and any other party being the receiver of such a transaction. However, because all legitimate CLZ transfers are necessarily validated by the Global zone, and take effect when epochs change, these honest parties already have ground truth validated by the Global zone (reason S1 Table 3.4). The latter point also touches on liveness guarantees: How long can honest senders and receivers be blocked by a misbehaving CLZ? The honest sender can simply transfer the CLZ again (reason L1), which takes effect and all parties observe when the Global zone validates the transfer (reason L2).

In case 3, the only compromised party is the old CLZ. The old CLZ could attempt to pretend

to not be the CLZ as a safety attack, making the querier believe it is stuck in a past epoch. However, anyone can detect the bogus claim if the CLZ cannot present contrary information signed by the Global zone specifying a new CLZ in the current epoch (reason S2). Regarding liveness, the old CLZ could refuse to execute the CLZ change. Then, the sender can escalate the situation to the Global zone (reason L1), as presented in the protocol. When the sender escalates the situation, the receiver (and all other honest parties) can eventually observe the sender's transaction via a different well-behaved CLZ; the CLZ transfer takes effect when the Global zone validates the transfer (L2).

In case 4, both the old and the new CLZ are compromised. We are interested in the guarantees for honest senders and honest actors. Crucially, the two CLZs cannot collude by skipping the Global zone, because any honest party can detect that the Global zone has not signed the transfer. Thus, the damage these two zones can inflict is simply the sum of the individual damage they can each inflict. We presented these in cases 2 and 3.

Compromised sender. Case 5 considers the sender to be the only compromised party. As a safety attack, a compromised sender can attempt a double CLZ transfer, however, the old CLZ is honest and would not sign conflicting transfers. Also, the compromised sender could pretend that the UTXO's CLZ is a different one and launch a transfer from there, but an honest zone would verify that it is not the CLZ and reject the transfer. A compromised zone might collude with the attacker and allow the transfer, but the Global zone would not sign the transaction.

As a liveness attack, the compromised sender could partially execute the transfer and then stop. For example, it could just have the transfer transaction in the source CLZ, but not in the Global zone or further. However, the Global zone periodically pulls transactions, so it eventually validates the transfer. Or, the compromised sender could have the transfer validated by the Global zone, but stop before informing the new CLZ. This would be a framing attack, because an honest party following the CLZ transfer pointer could believe that the new CLZ is malicious. However, the new CLZ eventually synchronizes with the Global zone after the long timeout, and would be able to complete CLZ transfers even without the sender's help (reason L2).

In case 6, both the sender and the new CLZ are compromised. They cannot, however, collude without being detected, because they need to involve the old CLZ and the Global zone, which are honest. Thus, the damage they can inflict is simply the sum of the individual damage the sender and the old CLZ each can inflict. We presented these in cases 2 and 5.

Case 7 considers the situation when both the sender and the old CLZ are compromised. The sender and the old CLZ could collude to allow double CLZ transfers: The old CLZ can sign one transfer, delete the history to conceal traces, and then sign another transfer. However, all zone transfers need to be verified by the Global zone, which would not approve this attack (reason S3). If the sender does not get a zone transfer approved by the Global zone, then an honest new CLZ does not accept the transfer.

Another unsafe situation is when the sender makes a zone transfer, the Global zones approves it, and then the old CLZ deletes the history to conceal the transfer. Those parties that contact the old CLZ and do not observe the CLZ transfer believe the old CLZ is still the rightful CLZ, whereas those that see the transfer believe the rightful CLZ is the new CLZ. The old CLZ could also create this situation in cases 3 and 4. However, here the sender is also compromised, and exploits the situation in order to double spend. This is especially problematic when the new CLZ is honest, as in this case, because it unwantingly helps with the double spending. Nyle detects this case because, after a CLZ transfer but before the new epoch, all transactions for that UTXO are processed by the Global zone (reason S3).

A colluding sender and old CLZ cannot prevent the liveness of the new CLZ any more than the sender and old CLZ can individually prevent liveness (cases 3 and 5). Also, these two colluding parties cannot hide forever the CLZ transfer: Before the transfer is observed by the Global zone, the transfer is in progress, and once the transfer is observed by the Global zone (which happens at least during periodic synchronization), the transfer completes (reason L2).

In the eighth and final case, the sender, old CLZ and new CLZ are all compromised and possibly colluding. We are interested in the guarantees for other honest parties. Regarding safety, all attacks discussed above are possible, and the honest party can counter them as discussed above (reasons S1, S2, S3). Additionally, the colluding parties could perform a CLZ transfer that bypasses the Global zone from the old CLZ to the compromised new CLZ. Then, the colluding source and old CLZ perform another transfer to an honest new CLZ, thereby attempting a double CLZ transfer. However, any honest actor can trivially detect the misbehavior, because the bogus transfer bypassing the Global zone does not have the Global zone signature (reason S4).

There is no additional liveness attack that the colluding parties can launch apart from the individual attacks in all cases above. Thus, an honest actor counters the liveness attacks like in the previous cases.

3.7 Preliminary results

In order to estimate the benefits and overheads of Nyle, we run a simulation that compares Nyle with Omniledger. The simulation has two components: the validator network and the transaction workload. The validator network takes a maximum exposure as input - we use RTT as the exposure metric - and generates coordinates for the nodes in a 2D coordinate space. Using the inter-node cartesian distance as the pairwise node RTT, the simulation creates the Nyle zones. Our network has 400 nodes with maximum RTT of 1 second.

For the Nyle zone radii, we use two settings. Nyle1 uses the same formula as Limix: $R_i = (i-2) * \sqrt{2}^i$, which reduces exposure predominantly for smaller zones. Nyle2 uses the formula $R_i = (i-2) * 2^i$, which is less aggressive in reducing exposure. Nyle uses the auto-zoning algorithm with a varying number of levels K. For the Omniledger setup, we follow the setup

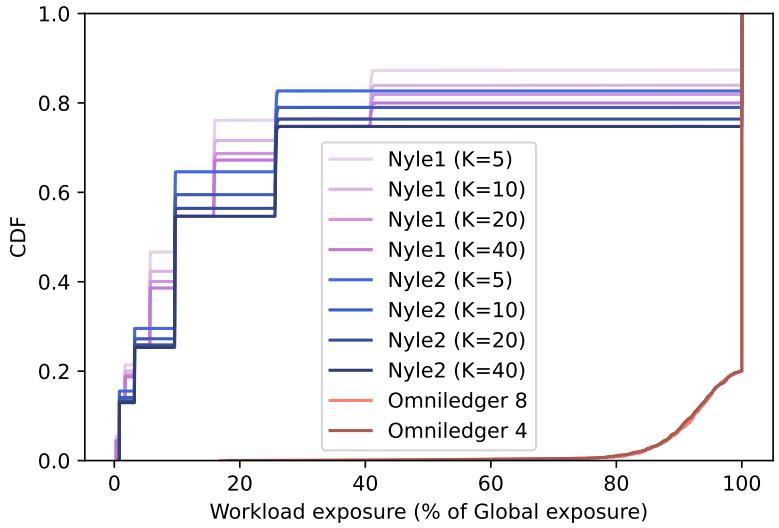


Figure 3.8: Simulation depicting a CDF of the workload exposure as percentage of a Global exposure. Nyle1 uses zone radii $(i-2)*\sqrt{2}^{i}$, and Nyle2 uses $(i-2)*2^{i}$. K represents the number of levels in Nyle auto-zoning. Omniledger uses micro-shards of 4 and 8 nodes.

used by the authors. We create shards of 80 nodes, and in each shard we create 20 micro-shards - each micro-shard has thus 4 nodes. We also run simulations with 10 micro-shards.

We run a workload of approx. 10'000 transactions, where each transaction has a sender and a receiver in the form of validators. The RTTs between the sender and the receiver follow the same RTT distribution as in the Limix workloads. This workload is representative for RTTs between Internet communication endpoints. Using real transaction histories, e.g., from Bitcoin or other cryptocurrencies, would be a potentially worthwhile area for further experimentation.

Exposure. Our simulation results compare Nyle and Omniledger on two dimensions: transaction exposure for the workload and the load on validators. For transaction exposure, we measure the RTT diameter of the zone/shard that optimistically validates and settles the transaction In Nyle this is the smallest zone of the sender and receiver; in Omniledger, it is their smallest shard or micro-shard, or the inter-shard latency for a cross-shard transaction. For both systems, we compute the reduction in exposure against a global ledger, where all transaction have a global exposure equal to the maximum inter-node RTT.

Fig. 3.8 depicts the exposure results. We observe that Nyle significantly reduces the workload exposure, with similar results across different level numbers K: 70% of the transactions observe an exposure in Nyle1 of only 20% compared to a Global exposure, respectively 30% in Nyle2. In contrast, only 20% of transactions in Omniledger observe any improvement in exposure, and the improvement is modest: the exposure is 80% or more of a Global exposure.

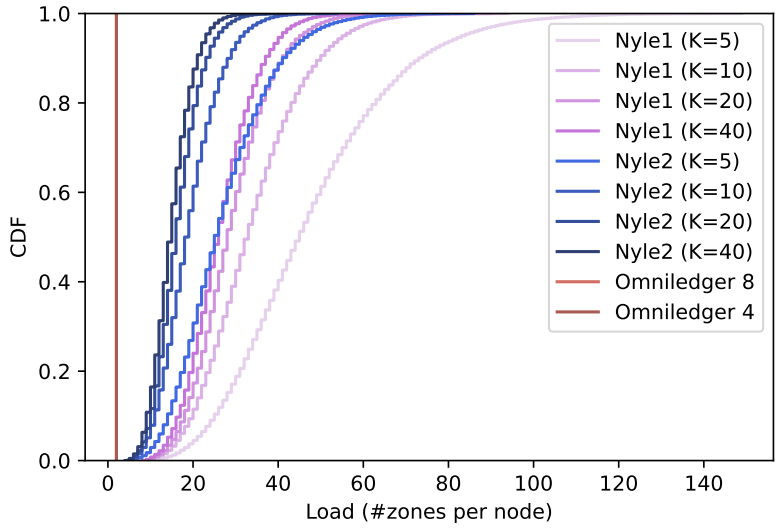


Figure 3.9: Simulation depicting a CDF of the load measured as the number of zones per validator. Nyle1 uses zone radii $(i-2)*\sqrt{2}^{i}$, and Nyle2 uses $(i-2)*2^{i}$. K represents the number of levels in Nyle auto-zoning. Omniledger uses micro-shards of 4 and 8 nodes.

Load. We also measure the load as the number of zones in which each node participates, and Fig. 3.9 depicts the results. The Nyle CDFs depict the price Nyle pays for the exposure reduction: 60% of the validators can participate in up to 50 zones in Nyle1 K=5. Nyle2, however, is less aggressive in reducing exposure, and the load is much lower than Nyle1: 60% of the validators participate in up to 30 zones for K=5, which drops to only 15 zones for K=40. Omniledger validators participate always participate in only 2 zones: one shard and one micro-shard.

We anticipate that Nyle will be able to amortize the overhead per zone through implementation. In order to disseminate transactions, ledgers typically use gossiping in their peer-to-peer network in the Global zone. Using this existing mechanism, smaller zones incur no extra overhead to obtain transactions. Transaction confirmation via signature aggregation would incur a linear overhead if implemented naïvely, because different zones have a different membership and thus different signature shares to create the zone signature. However, there might be techniques where the signature shares could be reused, lowering this overhead.

3.8 Related Work

In this section, we discuss related work on various aspects of blockchain design: asymmetric trust, prioritizing safety or liveness, state sharding, trust-but-verify architectures, and payment channels.

Asymmetric trust. Instead of one global trust assumption for all, some ledgers enable each user to choose which nodes the user trusts and form its own quorums of validators that need to agree on a transaction. Ripple [100] introduced a model where each user selects trusted validators to build its own Unique Node List. Ripple does require users to have an estimate of the total number of validators, and considers at most 20% of the nodes to be Byzantine at any time. Although initially Ripple claimed that a list overlap of > 20% between user lists is sufficient for safety and liveness, Chase and MacBrough [37] showed that Ripple may not be safe unless the overlap is > 90%, and may not be live unless the overlap is > 99%.

Stellar [82] adopts a similar philosophy of asymmetric trust, however, it does not constrain user's chosen quorum sizes, as Ripple does. Instead, Stellar introduces federates byzantine quorum systems, where the system-wide quorum emerges from the combined choices of individual nodes. Each node (user/validator) chooses a validators set where any threshold of nodes form a quorum slice, and adopts the decision of any slice where a threshold of nodes agree. Thus the user is guaranteed safety as long as its chosen slices are honest, and liveness if at least one of the slices is available. The advantage of stellar is that nodes that do not interact do not need to be aware of each other, and also do not require a quorum.

Stellar's argument for global agreement rests on trust transitivity via multiple quorums. It is, however, unclear whether users and validators grasp the implications of their quorum slices. The paper outlines cases of configurations with "dangerous or meaningless thresholds", and emphasizes the need to periodically collect all peers' configurations (slices) and examine quorum intersection. Disjoint quorums may indicate risky configurations. Stellar not only detects but also aims to prevent system runs with unsafe configurations, thus it alerts nodes when the network is nearing a critical state where risky configurations are imminent. Stellar's reliance on the "Internet hypothesis" for its convergence reasoning appears to result in Stellar experiencing the same delayed convergence as the Internet when routing configurations are stale If Stellar publishes more information about deployment experience, it will be interesting to observe whether nodes may use these warnings to avoid problematic setups in tandem.

Motivated by understanding the relationship between asymmetric trust and standard quorum systems, Cachin et al. [27] formalize asymmetric quorum systems. They derive conditions with respect to the intersection between the quorums each node chooses such that the resulting system satisfies consistency (safety) and availability (liveness). They also define *wise* and *naïve* nodes as those nodes that are right, respectively mistaken in their trust assumptions over their selected quorums. Finally, they generalize well known protocols to the notions of asymmetric quorum systems.

Asymmetric trust is at the core of Nyle, however, users have a more constrained set of choices than in the systems above in order to avoid misconfiguration. Users can choose whether to trust some pre-existing local quorums in addition to the always trustworthy global quorum. Because the system defines these quorums instead of the users, and all users and nodes in the system know these quorums, it is easy to fulfill the conditions of quorum overlap for global

consistency. Moreover, this guidance in quorum selection also distributes load in the system, preventing compromised users from launching denial of service attacks.

Prioritizing safety or liveness. While some ledgers choose to prioritize either safety (finality) or liveness (availability), others give the choice to clients. Neu et al. [93] refer to this issue as the finality-availability dilemma. The authors define the new ebb-and-flow protocols that make use of two ledgers: one ledger is available but transactions may not be safe/finalized, just as Bitcoin, and the other ledger is finalized but may not be available at all times, just as PBFT. The finalized chain is always a prefix of the available chain. Clients have the choice of which chain to follow for their transaction confirmations, depending on how conservative or aggressive clients are.

Similar to ebb-and-flow, Nyle provides clients a selection of ledgers, including a global safe ledger. However, in contrast to the ebb-and-flow protocol above, where both ledgers run a global consensus and hence have global exposure, Nyle has non-global ledgers running a localized consensus to reduce exposure and boost availability. Clients may choose which local ledgers to trust, if any. Moreover, all Nyle ledgers prioritize safety, thus a wise client is guaranteed safety and more likely to experience availability in a local ledger than in a global ledger. As with ebb-and-flow, naive or cautious clients may always rely on the safe global ledger.

State sharding. Some blockchains designs employ zones, just like Nyle, but they target scalability and do not bound exposure. Omniledger [76] and Monoxide [125] employ zones for scalability, where zones process transactions in parallel, thereby increasing throughput. Because each transaction is processed by a single zone, every zone needs to be secure in order to prevent double-spending, which Omniledger ensures by grouping in zones a large number of randomly selected nodes. However, nodes in the same zone could be located anywhere; random selection enhances security, but also increases Lamport exposure.

Trust but verify. In order to decrease transaction latency, some blockchains rely on the trust-but-verify model, which Nyle also adopts. However, they lack a notion of locality, which prevents them from reducing exposure. Omniledger [76] creates micro-zones within each shard that validate transaction faster than the shard because they require synchronization between fewer nodes. These micro-zones might not be secure, however, therefore each transaction is also validated by the secure shard. Because these micro-zones contain randomly selected nodes, they lack a locality notion, e.g., validators might not be in the same legal jurisdiction or even within a certain latency boundary. Thus, they do not bound exposure.

Payment channels. Off-chain approaches, such as payment channels [84, 96], bypass the main chain in order to increase throughput, but do not aim to nor reduce exposure. Individual users create on-chain deposits, and then initialize pairwise channels each capped by the

deposit either directly between users, or between users and payment hubs. Users transact with each other by sending signed transactions to the channel counterparty, and this step can repeat multiple times between channel counterparties to reach the destination. Every transaction cumulates the previous payments on the channel, so that when the channel is closed by the parties, they settle on the blockchain just a single transaction containing the final channel balance. Thanks to transaction summarization, channels increase throughput because the main blockchain processes fewer transaction.

Some payment channels seem to suffer from transaction latencies higher than for on-chain transactions, because channels require a transaction dispute period before the final on-chain settlement. Ethereum's feature of rollups [47] resembles channels, but attempts to address the latency issue. Rollups summarize transactions and have two modes of operation. Optimistic rolllups require a grace period, just like channels do, for disputes. However, zero-knowledge rollups eliminate the grace period by providing publicly verifiable proofs that the transaction summaries correctly represent the batched transactions.

Channels have some exposure-limiting property, however, it is merely temporary and ad-hoc. When parties settle transactions on a direct pairwise channel without intermediaries, then indeed their transaction has the ideal Lamport exposure. However, deposits need to replenish periodically, which takes place through an on-chain transaction with global exposure. Also, creating and maintaining direct user channels requires prohibitively large on-chain deposits. Therefore, most payments traverse payment hubs, leading to centralization and potentially problematic trust chains. With intermediate hubs, the end-to-end transaction is exposed to all intermediate parties clearing the payment, which they may refuse to do, leading to liveness problems. Or the intermediary parties might be located far away, increasing the exposure to potentially global. Most critically, users cannot decrease the exposure because they have no influence on the choice of intermediate hops.

3.9 Discussion

We discuss two exposure metrics introduced by Limix, and their relevance in the context of Nyle: geographical jurisdictions and RTTs. We also discuss possibilities for securing RTT-based zoning.

The first exposure metric are existing jurisdictional boundaries such as countries and economic zones, taken as input. This metric is advantageous because it supports the enforcement of existing laws. Transactions between parties within a jurisdiction are subject to that jurisdiction's laws and regulations, and the metric measures to which extent the transactions can be under the influence of parties outside the jurisdiction. For example, we can limit to what extent validators outside Germany decide on the validity of transactions between parties in Germany. A high exposure is undesirable, because it might not be possible to hold outsiders culpable under the jurisdiction's laws, even though they decide the validity of an intra-jurisdiction transaction. Such hurdles are a reality in Internet routing, where it is difficult

to hold a foreign ISP culpable of traffic delay. Also, the metric might support the enforcement of the travel rule, because a low exposure means that transacting parties are more likely to adhere to a common jurisdictional sanction base and aid local law enforcement to access transaction data [118].

The second exposure metric is the network round-trip time (RTT) between validators. Validators within smaller RTTs from each other communicate faster, and confirm transactions faster, thus the metric indicates an estimate of the transaction confirmation time. The advantage of this metric is that parties who watch a transaction, such as the receiver, can take informed decisions within the trust-but-verify architecture. For example, the receiver can decide in discrete intervals how long to wait for confirmations, and the security level and exposure of each confirmation.

It is worth pointing out that both metrics could result in multi-level zone hierarchies and non-strict zone hierarchies, where zones overlap partially. Multi-level hierarchies arise naturally, for example Germany is part of the EU, which is part of Europe. Also, an area might belong to multiple zones that do not fully overlap. For example Colorado City located in the United States crosses the state border between Arizona and Utah, and is thus located in both states; Pettigo is a city located in two countries: Northern Ireland and the Republic of Ireland; Baarle is a border town located in both Belgium and The Netherlands, with non-continuous pieces of land in either country; and the examples can continue. The RTT metric does not by itself produce such hierarchies, but it does so when used in compact graph approximations as in Limix.

Securing RTT-based zoning We described that Nyle enhances the Global zone membership by having each validator sign a verifiable location claim at registration time and add it to the membership blockchain. However, declaring location is insufficient when applying an exposure metric such as RTT. This is because RTT-based zoning requires inter-validator latencies, but RTTs generally do not correlate with geographical distances, albeit for small distances and some specific locations, e.g.,, well-connected metropolitan areas. Compromised nodes could arbitrarily increase their RTTs to benign nodes, potentially pushing benign nodes out of certain zones because these benign nodes seem “far from everyone”. To secure RTTs, we could leverage secure virtual coordinate systems that assign coordinates to nodes based on their pairwise RTTs [42, 104]. We also refer to a protocol developed by Kall et al. [70] for Nyle. Recently, Kohls and Diaz [74] proposed verifying geographical location claims via RTTs. This suggests that we could use the correlation between the two metrics to derive a potentially incomplete RTT map. However, measuring the entire RTT map is likely noisy with a high number of nodes. A remaining challenge are thus autozoning algorithms that can accommodate incomplete RTT maps.

3.10 Summary

This chapter described Nyle, a trust-but-verify distributed ledger architecture that enhances existing BFT-based blockchains with exposure-limiting capabilities for transactions. To limit exposure, Nyle uses similar zoning techniques as Limix, but additionally secures the exposure metrics reported by possibly Byzantine nodes. In Nyle, smaller local zones may be compromised and break safety or lose liveness. To ensure safety, the global zone always verifies all transactions and acts as a secure finalizer, whereas local zones act as opportunistic finalizers that each user may independently choose to trust or not. Nyle also addresses transaction censorship by compromised zones, and other liveness attacks. Preliminary simulation results show that Nyle significantly reduces exposure compared to Omniledger, with 60% of transactions witnessing 70% less exposure in Nyle, at a cost of 10x increase in load.

4 Asynchronous Consensus without Common Coins

We have shown in the previous chapters that it is possible to build globalized systems with strong consistency, where local interactions need just local coordination. While Limix shields local interactions from remote failures and slowdowns, the consensus protocol must nevertheless account for local inner-zone failures. In fact, the network in any zone may become asynchronous at any time due to various failures and network asynchrony within the locality, caused by partially functional equipment [81] or (temporary) misconfiguration [2, 68], or even with malicious intent by an adversary that applies selective censorship on network links or launches a DoS attack [1, 109, 111]. Hence, we would prefer consensus to operate as robustly as possible - in terms of both availability and performance - in the face of arbitrarily adverse network conditions, and not just in "normal-case" conditions. The asynchronous network model is the theoretical model that captures this maximally adverse scenario.

Most currently-deployed approaches to arbitrary value consensus impose network synchrony constraints. Arbitrary value consensus was introduced by Pease et al. [95], and many algorithms followed since using various assumptions. Some algorithms, such as Paxos [20, 78, 79] and its variations [20, 36, 86, 86, 90, 122], and newer protocols such as HotStuff [131], assume partial synchrony. However, synchronous and partially/weakly synchronous protocols may suffer from unnecessary delays in the presence of asynchrony, because they rely on timing assumptions [89]. In partially synchronous conditions, HotStuff drives consensus "at the speed of the actual (vs maximum) network delay" [131] - a property the authors call responsiveness. But, partially synchronous protocols may have infinite executions over an asynchronous network, meaning they cannot ensure liveness. Thus, arguably, they are not responsive under asynchrony.

Protocols that do ensure progress under asynchrony must rely on randomization to prevent executions from not terminating [51]. To this extent, asynchronous consensus protocols [9, 73, 83, 89, 114] usually require common coins, a primitive that offers the same random value to callers that is unpredictable to an adversary.

It is natural to ask whether common coins are required for efficient consensus. In this chapter,

we present a negative answer to this question by introducing Que Sera Consensus (QSC), a novel arbitrary value asynchronous consensus algorithm for crash faults. Other algorithms need (perfect) common coins to elect a leader or to pick the same value from a set of reliably broadcast values, neither of which QSC needs. Instead, QSC builds atop Threshold Synchronous Broadcast (TSB), a novel communication primitive, and private randomness. TSB enables the *step-synchronous* distribution of a *threshold* of values to all live nodes, unlike reliable broadcast, which distributes *all* values to all live nodes *eventually*. Nodes converge to the same value within the optimum expected $O(1)$ rounds and $O(n^2)$ message complexity ($O(n^3)$ communication complexity) by using private randomness tied to the values.

One could argue that QSC actually builds a common coin, however, that would be imprecise. Common coins select a random value from an input set, but are not concerned with obtaining the input set. Assuming that all caller nodes call the common coin with the same input set, the coin ensures that all nodes eventually select the same random value. Consensus is a strictly stronger primitive than common coins: Besides ensuring that all nodes agree on the same value eventually, consensus *additionally* guarantees validity, i.e., that the agreed value was proposed by some node. For validity, consensus needs to disperse the input values between the nodes. Observe that, even for binary consensus, nodes could not simply call a common coin with the input set $\{0,1\}$, because they might break validity if they decide e.g., 0, but nobody proposed 0. QSC builds consensus, which is strictly stronger than common coins, and it does so without relying on an existing common coin.

Asynchronous consensus protocols are rarely, if ever, deployed in practice. Our hope is that QSC is a first step towards more practical asynchronous consensus protocols. QSC removes the need for common coins in the crash fault setting. Also, TSB seems to intuitively fulfill a responsiveness property in asynchronous networks, because it enables group communication at the speed of a threshold of nodes. QSC also appears elegant and modular, although these properties are hard to quantify.

This chapter is organized as follows. We first give background into asynchronous consensus and review related work. Then we give the system model and the overview of QSC and FSTSB (full-spread TSB), a particular variant of TSB that QSC builds on. Next, we provide a detailed description of QSC, followed by proofs of consensus safety and liveness. Afterwards, we define TSB thresholds, properties, and describe a TSB implementation. Finally, we provide a discussion and summarize the chapter.

This chapter is based in part on joint work with Pasindu Tennage and Eleftherios Kokoris-Kogias, Philipp Jovanovic, Ewa Syta and Bryan Ford. Parts of this work appear in preprints available online [52, 53].

4.1 Background and Related Work

This chapter gives background information on consensus and compares related work with QSC's approach. We first review common network and failure models used in consensus, and the impossibility of deterministic asynchronous consensus. Then we look at building blocks for probabilistic asynchronous consensus, such as common coins. Finally, we survey probabilistic consensus algorithms and their complexity, and contrast them with the approach QSC takes. Although QSC operates with crash faults, we examine protocols with crash faults and Byzantine faults. Because every protocol that can withstand Byzantine faults can also withstand crash faults, we want to highlight the essential differences between QSC and these protocols.

4.1.1 Consensus background

Consensus is a fundamental problem in distributed computing where a set of n nodes each propose a value and need to agree on a value among these. Specifically, consensus satisfies the following properties [32]:

- *Agreement:* No two correct nodes decide differently.
- *Validity:* If a node decides v, then v is a value proposed by some node.
- *Integrity:* No correct node decides twice.
- *Termination:* Every correct node eventually decides some value.

Modelling nodes. The definition specifies correct nodes, which brings into discussion the abstraction we use for nodes. One popular abstraction that this chapter uses is *fail-silent*, where all nodes faithfully follow the protocol, but some may crash at some point during the execution of the protocol. Node crashes are silent, meaning no other entities are "notified" of the crash. All live nodes are correct, but once a node crashes, it becomes incorrect and remains crashed for the entire duration of the protocol. Another abstraction that some of the related work uses is *fail-arbitrary*, where failed nodes can deviate arbitrarily from the protocol, also known as Byzantine nodes. The fail-arbitrary model is more general than fail-silent, and any algorithm that works in the former model also works in the latter. Byzantine nodes can be under the control of a single adversary and collude in their attacks. We denote by f the maximum number of incorrect nodes.

Modelling communication. Another aspect implicit in the consensus properties is that the nodes need to communicate in order to agree, and one critical aspect concerns timing. Timing assumptions refer to whether there is a known or even bounded time that elapses between a node sending a message and the destination node receiving this message. Several timing assumptions are possible, such as synchrony, partial/weak synchrony or asynchrony. In this chapter, we consider the most generic model: asynchrony. Messages eventually reach the

destination, but they may take an arbitrarily long time to do so.

Deterministic asynchronous consensus is impossible. Because deterministic algorithms cannot guarantee termination (liveness) of asynchronous consensus [51], algorithms that do guarantee progress must rely on randomization to escape from unfavorable executions. Fischer-Lynch-Patterson [51] gave a renowned impossibility result. They asserted that asynchronous consensus is impossible even with a single failing process because there are executions when several proposals are equally probable outcomes. If such an execution is incapable of breaking the symmetry, it does not terminate. There are two alternatives for termination: failure detectors or randomization [15]. Failure detectors allow identifying and discarding failing nodes, which breaks the symmetry because there is no uncertainty: All nodes eventually receive the non-failing nodes' messages. However, instead of fail-silent nodes, failures detectors require fail-stop or fail-noisy nodes, which are less general. QSC chooses the alternative approach, which is to keep the general fail-silent model and break symmetry via randomization.

Message scheduling. Based on the previous paragraph, termination depends on which messages each node observes. The asynchronous network model generally assumes that messages may be scheduled adversarially, i.e., in the worst possible way for a consensus algorithm. In most asynchronous consensus algorithms that rely on randomization, including QSC, an adversarial network scheduler would be able to defeat the consensus algorithm if it could obtain the random coin flips "too early" and schedule messages accordingly. Therefore, like many asynchronous consensus algorithms, QSC assumes an adversarial but content-oblivious network scheduler [15]. TLS-encrypted private channels are a common way to keep the adversary content-oblivious in practice.

The first randomized consensus algorithm. Ben-Or [18] gave the first randomized algorithm for asynchronous consensus that terminates with probability 1. The algorithm achieves binary consensus, where nodes need to decide either 0 or 1. Nodes run the algorithm over multiple rounds, each consisting of two phases, where each node attempts to decide based on whether it observes that the majority of processes proposed the same value. If no decision is possible after the two phases, each node randomly picks 0 or 1 to propose for the next round. The algorithm has $O(n^2)$ communication complexity in the crash-fault model, but an exponential running time, because the algorithm is guaranteed to terminate only when a supermajority of nodes pick the same random value.

The algorithm above highlights a couple of challenges. First, it presents binary consensus, which is one possible flavor of consensus that can decide only 0 or 1. However, binary consensus can be insufficient for some distributed applications where nodes propose non-binary values [30]. Arbitrary value consensus is another flavor of consensus where the set of possible decisions is not known apriori. In both flavors of consensus nodes need to learn of each others'

values, but communication in arbitrary value consensus is more expensive, as we explain below.

The second challenge that it highlights is the cost of consensus. The community aims for an expected constant running time ($O(1)$) and a $O(n^2)$ or better communication complexity. The communication complexity includes number of exchanged messages and message size.

The algorithm of Ben-Or [18] described above uses private randomness, where each node independently chooses a random bit that it never sends to other processes. However, approaches exist for nodes to obtain the same random value more efficiently, and these are called *common coins*, which we describe next.

4.1.2 Common coins

Intuitively, a common coin offers the same random value to callers that is unpredictable to an adversary. Common coin is an abstraction introduced by Bracha [21]. The coin selects the value uniformly at random from a set given as input, for example $\{0,1\}$ for a binary common coin. If the coin always outputs the same value across all callers, it is called perfect; otherwise, it is biased, and the number of rounds it takes for the coin to give perfect outputs gives the coin's time complexity. In consensus executions where nodes cannot decide, if nodes keep proposing the same values, the undecidable execution might repeat infinitely. Common coins break this symmetry by offering nodes a valid value to propose that is likely the same for all nodes, hence the nodes are more likely to meet conditions favorable to deciding.

Common coins in Byzantine binary consensus. Although QSC targets a non-Byzantine model, it is still instructive to examine briefly the literature in common coin protocols for Byzantine consensus. Common coins used as a building block for consensus affect the practicality of consensus. All the coins described here defend against an adaptive adversary. In order to obtain constant-time complexity for consensus, one needs either a perfect practical common coin, or a common coin with constant bias, which requires a constant number of rounds to output the same value for all caller nodes. The first perfect common coin required predistributed information by a trusted dealer [98], which is a strong assumption because a compromised dealer breaks the entire system. Using this coin, Mostéfaoui et al. [91] provide a constant-time binary agreement protocol that avoids signatures by using authenticated channels, with a message complexity of $|m| * O(n^2)$, where $|m|$ is the message size. Later work still used a trusted dealer, but only for setup: Cachin et al. [29, 31] build a constant-time common coin using threshold cryptography with $l * O(n^2)$ message complexity, where l is the size of signatures in the message. Cachin et al. [30] use this coin to design a binary agreement protocol with $O(1)$ time complexity and $l * |m| * O(n^2)$ message complexity, and SINTRA [26] provides an implementation.

Some protocols used to bootstrap common coins remove the assumption of a trusted dealer. Canetti and Rabin [33] removed the trusted dealer restriction with their constant-time com-

mon coin with $O(n^2)$ total communication complexity. However, their communication protocol uses very large constants, which is prohibitive in practice [29, 91]. The most efficient recent asynchronous common coins have a communication complexity of $O(n^3)$ in total, and either $O(1)$ [43] or $O(\log n)$ time complexity [10], but require complex distributed key generation protocols.

Common coins in crash-silent binary consensus. Attiya and Welch [16] adapt the coin of Canetti and Rabin [33] for the crash-silent model with an oblivious adversary. The authors introduce the concept of *common core*, which is a set of $n - f$ values that all nodes are guaranteed to receive after engaging in three rounds of broadcast with $O(n^3)$ communication complexity. In fact, the nodes obtain a superset of the common core – obtaining the exact same set would be equivalent to solving consensus –, but cannot identify which of the values are actually part of the common core. The coin has constant bias, thus expected $O(1)$ running time, and a bit complexity of $O(n^3)$. Using the coin and the concept of common core, the authors propose an approach for binary consensus that uses majority voting, and obtain the same complexity as the coin. More recent protocols [114] assume a shared secret among the nodes, distributed by a trusted party, in order to bootstrap a perfect common coin.

Like Attiya and Welch [16], QSC's novel group communication protocol TSB delivers a common core to every node without needing a trusted dealer. However, in contrast to their work, TSB also enables each node to identify some non-empty subset of the values that definitely belong to the common core, which is crucial for QSC's safety properties.

4.1.3 Arbitrary value consensus

Arbitrary value consensus is a more difficult problem than binary consensus, because it solves agreement on a value from a set not known apriori. Binary common coins know the domain apriori and output 0 or 1. In contrast, choosing a random proposal in arbitrary value consensus requires knowledge of some node proposals in order to satisfy validity. In binary consensus, nodes broadcast values in order to fulfil validity, and use the majoritarian value to converge on the decision. In contrast, arbitrary value consensus cannot rely on majorities because the domain of values is larger.

Approaches using reliable broadcast. In the literature, there are two directions to arbitrary value consensus. One direction is to use *reliable broadcast*. Simply using n instances of reliable broadcast merely guarantees that eventually all nodes receive the same set of values, however, the nodes cannot tell when the set is complete. Cachin et al. [32] describe a *crash-silent* algorithm that, at the end of each round, feeds the possibly incomplete reliable broadcast set into a common coin. Eventually, when the sets are complete, the coin has the same domain on all nodes and outputs the same random value, leading to an $O(N^3)$ bit complexity and $O(N)$ expected time.

Miller et al. [89] and Keidar et al. [73] take the approach of amortizing the cost of reliable broadcast by agreeing on multiple values per consensus instance. Their proposals solve Byzantine consensus with an adaptive adversary. Both use the construction of Cachin et al. [28] to instantiate reliable broadcast with a communication complexity of $O(N|l| + N^2 \log N)$, where $|l|$ is the message size, and they batch $O(N \log N)$ values without increasing the communication complexity. Honeybadger [89] proposes a construction for an asynchronous common subset (ACS) (original proposed by Ben-Or [18] for a different purpose), where nodes run n parallel reliable broadcasts and input 1 into n corresponding binary agreement instances of Mostéfaoui et al. [91] to signal the completion of a reliable broadcast instance. When $n - f$ of the consensus instances complete, the protocol stop. Thus, their protocol agrees on a set of values. The communication complexity is $O(N^3 \log N)$ (amortized $O(N)$) and the running time is $O(log N)$ in expectation (but can be reduced to $O(1)$ at the expense of throughput). Keidar et al. [73] obtain the same (amortized) communication complexity and $O(1)$ running time in expectation.

QSC does not rely on reliable broadcast, but instead introduces TSB, a communication primitive ensuring that, when the call returns, each node receives a common core of values and can identify a non-empty subset of values that are part of the common core.

Consensus approaches using leader election. Other Byzantine consensus protocols take almost the opposite direction: There is no need to reliably broadcast all values when we only require agreement on a single value. Thus, they reduce the communication cost by electing a leader (via a common coin) as early as possible and only concern themselves with agreeing on the leader's proposal. VABA [9] is the first protocol to achieve $O(n^2)$ communication complexity and $O(1)$ expected termination. VABA first ensures that a node can become a leader only if its value has been received by sufficient nodes. It uses four rounds of a secure echo-broadcast, each of which ensures that $f + 1$ honest parties observed the broadcasted value and giving increasingly stronger guarantees about which nodes observed certain values. Each node that managed to promote its proposal nominates itself for a leader election phase. From that point, nodes only concern themselves with what other nodes observed about the leader's proposal and either decide, or run a view change that ensures safety across rounds.

Dumbo-MVBA [83] also uses leader election via a common coin, however, for large messages of size $O(n)$. Their contribution is to reduce the communication complexity for broadcasting proposals from $O(l * n^2)$ to $O(l * n)$ for broadcasting n proposals of size l. They do so by having each node threshold-share its proposal, which takes per node n messages of size $O(l/n)$. The coin tossing has a communication complexity of $O(n^2)$, which dominates the cost, however, it does not depend on the message size. Thus, Dumbo-MVBA can reach consensus on messages of size $l = O(n)$ for a total communication complexity of $O(n^2)$ and expected termination in $O(1)$.

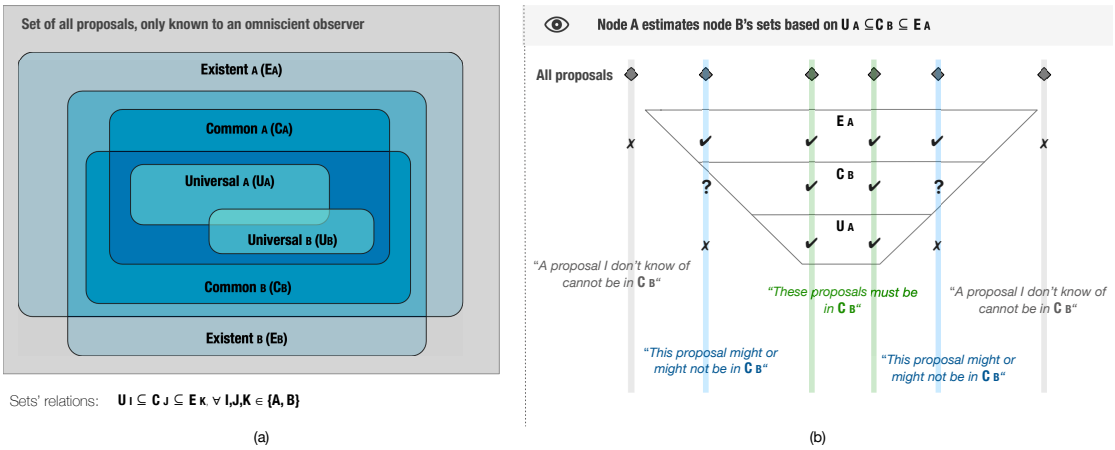


Figure 4.1: Illustrative example of the QSC sets of two nodes A and B. (a) The Existent set of each node represents the node's view of proposals; the Common set represents the node's view of proposals in all live nodes' Existent sets; and the Universal set represents the node's view of proposals in all live nodes' Common sets. The respective sets of different nodes may not be the same. The guarantee is that $U_I \subseteq C_J \subseteq E_K, \forall I, J, K \in \{A, B\}$. (b) Node A's knowledge based on its own sets Existent_A and Universal_A of node B's Common_B set.

Comparison with QSC**.** Although QSC is an approach for crash-silent but not Byzantine arbitrary value consensus, we believe it is worth contrasting QSC to the approaches above to give an intuition. Like the approaches above, QSC avoids broadcasting sets of values of size $O(n)$, in order to keep the communication complexity low. However, unlike the approaches above, QSC does not achieve that by electing a leader or using a common coin. Instead, QSC relies on trimming the set of values that could be decided on in two steps per round. Because nodes may not always trim the same values, several decidable values may survive, and nodes in QSC converge towards the same value using private randomness attached to the values.

4.2 Overview

QSC is an approach to achieving arbitrary value consensus in the fail-silent model. This section explains how QSC achieves consensus without depending on common coins or a setup dealer. QSC enables every node to estimate (limit uncertainty on) the values that every other node is aware of. We first give the intuition of why QSC's approach of limiting uncertainty benefits safety (validity and agreement). Then, we explain why TSB's thresholds and private randomness are essential for expected constant-time agreement.

4.2.1 Safety by limiting uncertainty

Challenge 1: Validity. Consensus protocols need to ensure consensus validity. Validity demands that nodes decide on a value proposed by some node; hence, nodes must be aware of each other's proposed values. Some protocols use reliable broadcast at each node, which

ensures that all live nodes eventually get the broadcast values. Due to their communication complexity, however, these approaches are only practical for batch decisions, but not for single-shot consensus.

Other protocols do not strive for the transmission of all values reliably; rather, they wish to be able to retrieve the value of any live node on demand. Each node repeatedly broadcasts its value to a quorum of other nodes using echo-broadcast, allowing anybody to obtain the value by later asking a quorum. Echo-broadcast has a lesser complexity of communication than reliable broadcast, making it suitable for single-shot consensus on small or big values.

Unlike the techniques above, QSC seeks to ensure that each node not only learns certain values, but can also limit the uncertainty between the values it learned and the values any other node may have learnt (Fig. 4.1-(a)). QSC accomplishes this by building on TSB, a new primitive for group communication that utilizes echo broadcast. Through 2 calls to TSB, each node i learns of three sets: an existent set E_i, a common set C_i, and an universal set U_i. The essential guarantee is the set inclusion $U_i \subseteq C_j \subseteq E_k, \forall$ nodes i, j, k. From the viewpoint of node i, its sets E and U set boundaries for the values in the C set of any other node. TSB helps QSC in ensuring validity, and limiting uncertainty is a stepping stone towards agreement, as we show in the next section.

Challenge 2: Agreement. Most existing approaches achieve agreement via the use of a common coin with a constant bias. Some systems, for instance, run binary agreement subprotocols that utilize a binary common coin. Others use a common coin to elect a leader, by providing the set of nodes as the coin input. All nodes elect the same leader, and then retrieve the leader's proposal from a quorum of nodes. However, constant-bias common coins need a dealer-operated setup step, an assumption that QSC eliminates.

QSC does not need a common coin, just private randomness. Recall that, from node $i's$ perspective, a proposal $P \in E_i$ and $P \in U_i$ must also $P \in C_j$, because $U_i \subseteq C_j \subseteq E_i$, $\forall$ i, j. However, there might be multiple such proposals P, as it is the case in Fig. 4.1-(b). To guide proposal selection, each node generates a random number (which we refer to as priority) and appends it to its proposal before broadcasting it using TSB. The intuition is that priorities create an ordering relationship across proposals, which nodes use to converge on their proposal selection across rounds. If the highest priority proposal $P_{Hi} \in E_i$ also satisfies $P_{Hi} \in U_i$, then the highest priority proposal in any other node's set C_j can only be P_{Hi} (our protocol also allows for priority ties). A node that observes this condition can decide. A node that cannot decide still relies on ordering to select its proposal for the next round. Section 4.3 describes the protocol in detail.

Note that priorities do not have a qualitative meaning regarding a proposal, In other words, no proposal is better than any other. The only goal of priorities is to speed up convergence, by enabling nodes to compare proposals and converge on the same one.

4.2.2 Liveness using random priorities

Challenge 3: Termination. Existing approaches that use a common coin depend on the coin's constant bias for optimal termination. In particular, it takes an expected constant number of rounds for the constant-bias coin to output the same value, given the same input.

QSC relies on the TSB thresholds for termination. TSB is a novel group broadcast abstraction that provides *threshold-reliability*, and consequently the E, C and U sets each contain a threshold number of values. QSC's termination is contingent on the probability that the highest priority value in E also exists in U, which is constant assuming adequate TSB thresholds (Section 4.4) and a network-oblivious adversary.

4.3 QSC: Que Sera Consensus

This section explains QSC in depth. First we present the system model and assumptions. Next we describe the QSC algorithm, which is based on private randomness and full-spread TSB. FSTSB is a special case of TSB, and we briefly outline how its characteristics determine the relationship between the E, C and U sets. Finally, we use the features of FSTSB and private randomness to argue for the safety and liveness of QSC.

4.3.1 System model

A group of n nodes run QSC to reach consensus. Of these, at most a quorum f can crash in the fail-silent model Section 4.1.1, with $n = 2f + 1$. Nodes that crash remain crashed forever and do not take any other step in the protocol.

The nodes are connected through a communication network. The network is asynchronous, which means that the network may take arbitrarily long to deliver any message, but it eventually delivers all messages between non-crashed nodes. The network can achieve that by using perfect links [32], implemented for example in TCP [113]. For simplicity, a node broadcasting a message sends n identical messages, one to each node including itself, which takes $O(1)$ time. In practice, one may also use broadcast algorithms that have a lower communication complexity at the cost of higher time complexity, such as protocols based on gossiping [48, 71]. However, using these techniques is outside the scope of this thesis.

We also assume that the network adversary is content-oblivious, which means that it cannot observe the contents of messages sent on links, nor any node state Section 4.1.1. In practice, this assumption can be realized by having nodes communicate over private channels (e.g., encrypted with TLS [99]).

Interface 7: Full-spread TSB (FSTSB) at node A.	
Call	: $R_A, B_A \leftarrow \mathsf{FSTSB}(m)$
Input	: m represents the message to be broadcast by the node
Output	: Sets R_A, B_A containing messages broadcast by nodes.
Parameters	: t_b represents the broadcast threshold, with $t_b > 0$ and $t_b \leq n - f$.
Property	: $\lvert B_A \rvert \geq t_b$
	: The messages in B_A are received by all live nodes at the end of the step: $\forall$ live node I, $B_A \subseteq R_I$.

4.3.2 The QSC algorithm

QSC builds atop two primitives: (1) a novel full-spread threshold synchronous broadcast (FSTSB) network abstraction, and (2) private randomness. We first briefly summarize their properties and then the QSC protocol.

Full-spread TSB (FSTSB). We briefly summarize full-spread TSB or FSTSB (Fig. 4.2), because is a fundamental component of QSC. TSB is a novel group broadcast abstraction that provides *threshold-reliability* regarding the messages broadcast and received by nodes. When a group of nodes uses the TSB primitive to broadcast a message each, and there are enough participants to meet certain thresholds, the call returns two message sets R and B to each node. R holds the messages received by the node; B comprises some messages that were each received by a threshold number of nodes.

FSTSB is a special case of TSB using certain thresholds to guarantee that all messages in any node's B set are received by all live nodes in their respective R sets. In other words, $B_i \subseteq R_j, \forall$ nodes i, j. Interface 7 depicts the primitive's interface.

By focusing on the communication properties for a threshold of the nodes, FSTSB enables slow nodes to skip FSTSB steps and catch up to quicker nodes performing subsequent FSTSB steps. It is essential to note that the R and B outputs of a node are not exclusive to that node. In fact, FSTSB (and TSB) does not guarantee, for example, that a node's own message appears in its own R and B sets. Consequently, a node may use the R and B sets of another node, and this would be a valid output satisfying FSTSB's properties. Section 4.4.4 provides further information on the "catch-up" process. In QSC, nodes repeatedly call FSTSB, as explained later in this section, and QSC may employ the FSTSB catch-up to allow slow nodes to catch up to subsequent QSC rounds.

Private randomness. We abstract private randomness through a node-private PrivateCoin primitive that operates independently from other nodes. At each call, PrivateCoin outputs a value selected uniformly at random from a domain D that has a partial ordering relationship $\preceq$. The domain D is publicly known. For example, D could be the domain of 32-bit integers, which has the ordering relationship $\leq$, and PrivateCoin calls the random number generator provided standard in many operating systems.

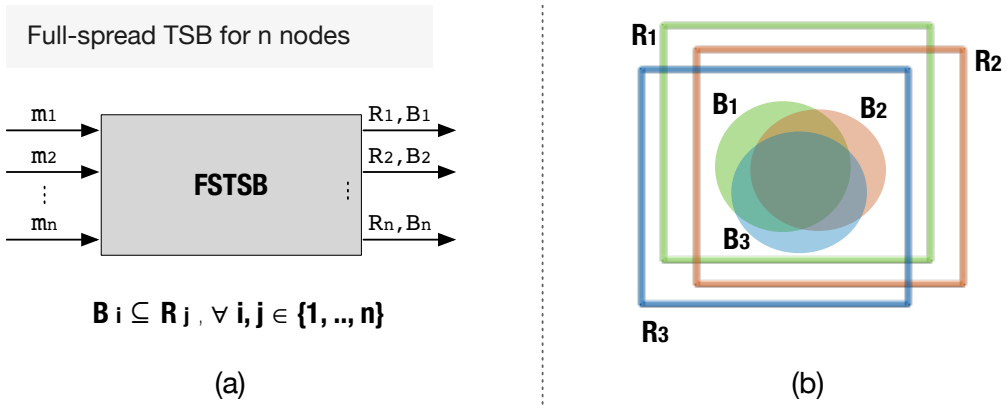


Figure 4.2: Full-spread TSB (FSTSB). (a) FSTSB is a group communication primitive, where each node inputs a message and outputs two sets R (received messages) and B (messages received by other nodes). The respective sets of different nodes may not be the same. The guarantee is that $B_i \subseteq R_J \forall I, J, K \in \{1, .., n\}$. (b) Illustrative example of the R and B set relationship for three nodes.

QSC. Algorithm 8 depicts the QSC algorithm. QSC proceeds in rounds, and each node starts a round with a proposal: either initial or chosen at the end of the previous round. The node appends to the proposal a random priority obtained using its PrivateCoin, and broadcasts it via FSTSB. The results of the first FSTSB execution initialize the sets Existent_A and B_A. The node then broadcasts the greatest priority (designated *best* for simplicity) proposal in B_A in the second FSTSB run, and initializes its sets Common_A and Universal_A with the execution results. The node decides if the best proposal in Universal_A is also the best proposal in Existent_A, and there is no other proposal with the same priority in either Universal_A or in Existent_A. Whether or not the node decides, it continues the protocol by choosing any best proposal from Common_A as its proposal for the following round.

Intuition behind the decision condition. We now show why proposal ordering by priority and the set inclusion relationship $\text{Universal}_I \subseteq \text{Common}_I \subseteq \text{Universal}_K, \forall$ nodes I, J, K, are crucial for safety. The proofs can be found in the next section. We modify the decision condition, and provide examples of how nodes that cannot decide may make unsafe choices. Assume that node A decides on some p_{Hi} that is not uniquely best in Universal_A. We take the perspective of another node I to explain all possible "undecided" cases, depicted in Fig. 4.3.

First, if A decided on $p_{Hi} \in \text{Existent}_A \setminus \text{Common}_A$, there can be a node I for which $p_{Hi} \notin \text{Existent}_I$ (Fig. 4.3-(a)). If none of the nodes are aware that p_{Hi} exists, they have no chance of ever selecting or deciding on p_{Hi}.

Second, if A decided on $p_{Hi} \in \text{Common}_A \setminus \text{Universal}_A$, we know that all live nodes I *must* have $p_{Hi} \in \text{Existent}_A$ due to the set inclusion relationship. However, how could these nodes determine which proposal is p_{Hi}? If such a node I merely selected the best proposal from its Existent_I set, it would not be safe, since it may choose a proposal P that A had not seen,

Algorithm 8: Que Sera Consensus (QSC) at node A.

Uses : PrivateCoin, FSTSB, $t_b = n - f$ unless otherwise specified
Input : v, the proposed value of node A
Output: v', the decided value

```
forever                                                    // loop forever over consensus rounds
    Existent_A, B_A, Common_A, Universal_A ← {}, {}, {}, {}        // Reset sets every round
    r ← PrivateCoin()                          // choose proposal priority using private randomness
    m ← v || r                                                  // append priority to proposal
    (Existent_A, B_A) ← FSTSB(m, t_b)          // broadcast my proposal and learn existent values
    m' ← any best proposal in B_A              // choose the best proposal that I know all nodes know of
    if m'.v is the value of all proposals in Existent_A then   // if all proposals have the same value
        if not decided then                                     // decide if not already decided
            Decide (m'.v)
        break                                                   // exit the algorithm
    (Common_A, Universal_A) ← FSTSB(m', t_b)   // broadcast m' and learn existent, universal sets
    m'' ← best proposal in Universal_A                          // choose any best universal proposal
    if m'' uniquely best in Universal_A then                    // proposal m'' is the only best
        if m'' is uniquely best in Existent_A then              // proposal m'' has no possible competition
            if not decided then                                 // decide if not already decided, but continue
                Decide (m''.v)
    m ← any best proposal in Common_A                           // continue with the best common proposal
    v ← m.v
```

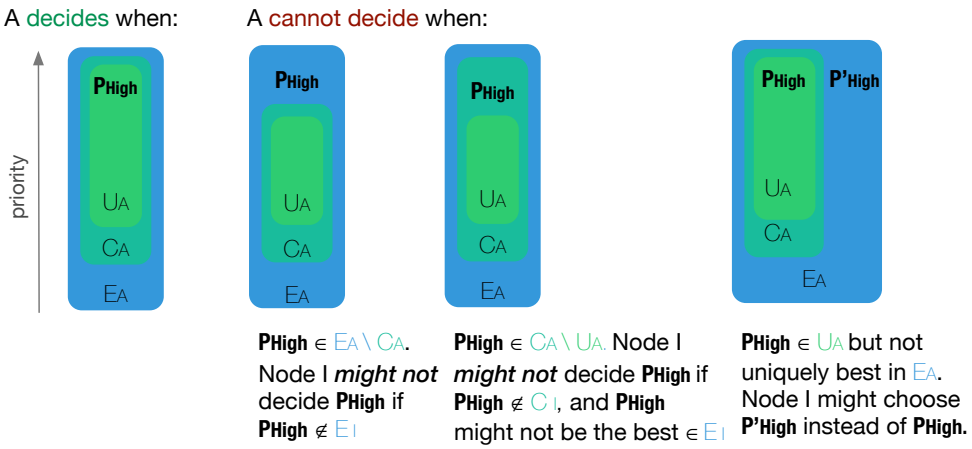


Figure 4.3: QSC decision process at the end of a round.

as in the previous example ($P \notin \text{Existent}_A$). And, because there may be nodes I for which $p_{Hi} \notin \text{Common}_I$, selecting the best proposal in Common_I would not safe either (Fig. 4.3-(b)).

Finally, if A decided on $p_{Hi} \in \text{Universal}_A$ but $\exists p'_{Hi} \in \text{Existent}_A$ with the same priority as p_{Hi}, then all other nodes I may have both p'_{Hi} and $p_{Hi} \in \text{Common}_I$ and may all choose P'_{Hi} for the next round (Fig. 4.3, (c))). In all of these situations, nodes cannot tell whether a proposal choice is safe.

Intuition for safety across rounds. If a node A cannot decide, it must choose a safe proposal for the subsequent round. A safe proposal must consider the possibility of a decision in the

present round. If another node I has already decided p, then all remaining nodes must also eventually decide p. Thus, the question is: is it unsafe if some nodes pick values different than p for the next round?

We argue that an "undecided" node A must select the best proposal $p_{Hi} \in \mathsf{Common}_A$ for the following round. Intuitively, any proposal p that a node I decided *must be the best* $\in \mathsf{Common}_A$. First, $P \in \mathsf{Common}_A$ since $\mathsf{Universal}_I \subseteq \mathsf{Common}_A$ and any node I that decides chooses its decision from $\mathsf{Universal}_I$. Second, node I that decided ensured that there was no better proposal $\in \mathsf{Existent}_I$. Consequently, because $\mathsf{Common}_A \subseteq \mathsf{Existent}_I$, there cannot be a proposal $\in \mathsf{Common}_A$ that is better than p, $\forall$ live nodes A.

A remaining question is: is it unsafe if a node chooses values different than the best in its Common set for the next round? The answer is yes because, depending on the priorities and network scheduling, certain nodes might decide on these unsafe values. If node A chose for the following round $p_{Hi} \in \mathsf{Existent}_A$, it is possible that $p_{Hi} \notin \mathsf{Existent}_I$ (Fig. 4.1). In other words, the node that decided may not be aware of p_{Hi}. The crucial observation is that all undecided nodes may behave like A. In fact, P_{Hi} may be $\in \mathsf{Existent}_A$ for all live nodes A, but still $p_{Hi} \notin \mathsf{Existent}_I$. Then, all undecided nodes choose p_{Hi} for the next round, and, although decided nodes choose p, there is a risk that the remaining nodes decide p_{Hi}. Hence, node A may violate safety by making an unsafe proposal in the next round.

Assume node A selects for the next round $p_{Hi} \in \mathsf{Universal}_A$. It is possible, however, that I's decided proposal $p \in \mathsf{Common}_A \setminus \mathsf{Universal}_A$. In other words, p is an universal proposal to an external observer and to some nodes, but node A is unaware of this fact. Consequently, A and other live nodes may introduce an unsafe proposal in the next round.

Liveness intuition. For liveness, QSC relies on two techniques: randomization of proposals and a content-oblivious message scheduler. (1) Randomization is necessary in QSC, as it is in all asynchronous consensus protocols, to avoid repeated "unfavorable" executions in which no node observes a decision condition. QSC ensures this by requiring each node to assign a fresh random priority to its proposal at the start of each QSC round. The node attaches a fresh priority regardless of whether it retained or changed its proposal. (2) QSC must prevent a network attacker from reducing the effect of randomness by timing message delivery such that no node observes a decision condition. Due to our assumption of a content-oblivious adversary, who cannot observe node state or message contents on network links, the adversary lacks the knowledge necessary to conduct an attack more effective than random message delivery.

4.3.3 Safety

This section provides proofs that QSC satisfies the validity, agreement and integrity properties of consensus.

Lemma 4.3.1 (Set relationship)**.** *In any* QSC *round,* $\mathsf{Universal}_I \subseteq \mathsf{Common}_J \subseteq \mathsf{Existent}_K$, $\forall$ *nodes* I, J, K *that complete the second* FSTSB *call of that round.*

Proof. Let node A be a node that completes its second FSTSB call in round r. We make two observations. By FSTSB's properties, $\mathsf{Universal}_A \subseteq \mathsf{Common}_I$, $\forall$ nodes I that complete their second FSTSB call in round r.

Also by FSTSB properties, all messages received in Common_A were broadcast by some node in the second FSTSB instance. We now show that the message m' broadcast by any node A in the second FSTSB instance has the property $m' \in \mathsf{Existent}_I$, $\forall$ nodes I that complete their first FSTSB call. Let node A be a node that completes its first FSTSB call in round r. By FSTSB's properties, $B_A \subseteq \mathsf{Existent}_I$, $\forall$ nodes I that complete their first FSTSB call in round r. Node A chooses $m' \in B_A$, thus $m' \in \mathsf{Existent}_I$, $\forall$ nodes I that complete their first FSTSB call in round r. This observation enables us to conclude that all messages in Common_A are also in $\mathsf{Existent}_I$, $\forall$ nodes I that complete their first FSTSB call; thus, $\mathsf{Common}_A \subseteq \mathsf{Existent}_I$.

Node A was chosen to be any node that completes its second FSTSB call in round r, and round r was not chosen in any special way, thus the relationship holds for any round. Together with the first observation, we conclude that $\mathsf{Universal}_I \subseteq \mathsf{Common}_J \subseteq \mathsf{Existent}_K$, $\forall$ nodes I, J, K that complete the second FSTSB call of any round r. □

Lemma 4.3.2 (Validity)**.** *In* QSC*, any decided value was a value proposed by some node.*

Proof. Let node A decide in some round r. We show that, after the first FSTSB call of any round r, all sets of node A $\mathsf{Existent}_A, B_A, \mathsf{Common}_A, \mathsf{Universal}_A$ contain only node proposals. Because node A decides either on a value from $m' \in B_A$, or on a value $m'' \in \mathsf{Universal}_A$, it follows that A can decide only on proposed values, regardless the round r when it decides. Since node A was not chosen in a special way, it follows that all nodes that decide decide on a proposed value.

We show by induction that all messages in $\mathsf{Existent}_A$ after the first FSTSB call of any round were proposed by some node. **Base case:** In round $r = 1$, every node executing the first FSTSB call broadcasts the input proposal and the set $\mathsf{Existent}_A$ is empty before the call. By FSTSB's properties, because all messages in $\mathsf{Existent}_A$ were broadcast by nodes during the first FSTSB call, all messages in $\mathsf{Existent}_A$ were proposed by some node. **Induction step:** Assuming that in round r all messages in $\mathsf{Existent}_A$ were proposed by some node, we show the hypothesis holds for round $r+1$. By Lemma 4.3.1, at the end of round r we have $\mathsf{Common}_A \subseteq \mathsf{Existent}_A$, and nodes continue to the next round with $m \in \mathsf{Common}_A$, so by the induction step $m \in \mathsf{Existent}_A$ must have been proposed by some node. Thus in round $r+1$, all nodes broadcast such a proposal via the first FSTSB, so all messages in $\mathsf{Existent}_A$ were proposed by some node.

By Lemma 4.3.1, after the second FSTSB call of round r we have $\mathsf{Universal}_A \subseteq \mathsf{Common}_A \subseteq \mathsf{Existent}_A$. Together with the claim above, we conclude that all values in $\mathsf{Universal}_A$ and

Common_A were proposed by some node.

Leveraging the properties of FSTSB, after the first FSTSB call of every round $B_A \subseteq \mathsf{Existent}_A$, thus all values in B_A were also proposed by some node. □

Lemma 4.3.3. *Integrity. No node decides twice.*

Proof. A QSC node may decide only in two cases: via the early decision path or via the normal decision path. Both path check whether the node decided already, and only decide if the node had not already decided. □

Agreement. There are two conditions in QSC where processes can decide. When a node A decides using the first condition, i.e., when all proposals in the $\mathsf{Existent}_A$ set have the same value, we say that it takes an *early path decision.* When the node decides using the second condition, i.e., when the best proposal in $\mathsf{Universal}_A$ is uniquely best in both $\mathsf{Universal}_A$ and $\mathsf{Existent}_A$, we say that it takes a *normal path decision.* We prove lemmas for both decision paths.

Lemma 4.3.4 (Early path decision.)**.** *Assume w.l.o.g. that node A is the first node to decide. Let node A decide v in round r via the normal path decision. Then no node can decide $v' \neq v$ in round $r' \geq r$.*

Proof. We prove by contradiction. Assume ∃ node I that in round r' decides $v' \neq v$. We analyze separately the case when $r' = r$ and $r' > r$.

By the early decision condition of node A, $\forall p \in \mathsf{Existent}_A$, $p.v = v$. From the FSTSB properties, ∀ node I that completes the first FSTSB instance in round r, $B_I \subseteq \mathsf{Existent}_A$. Thus, it must be that $\forall p \in B_I$, $p.v = v$, and the $m' \in B_I$ selected by node I has $m'.v = v$. Our first conclusion is that, in round $r' = r$, no node can decide via the early path on $v' \neq v$.

By Lemma 4.3.1, $\mathsf{Universal}_I \subseteq \mathsf{Existent}_A$ in round r, and if node I decides $v' \neq v$, then $\exists p' \in \mathsf{Existent}_A s.t. p'.v = v'$, which contradicts our hypothesis that in round $r' = r$ node I decides $v' \neq v$.

Assume ∃ node I that in round $r' > r$ decides $v' \neq v$. ∀ node I advancing to round $r+1$, node I proposes any best $p' \in \mathsf{Common}_I$ of round r. By Lemma 4.3.1, $\mathsf{Common}_I \subseteq \mathsf{Existent}_A$ in round r, thus it must be that $p'.v = v$. Because the sets are reset at the beginning of round $r+1$, and all proposals have value v, no node can ever decide any other value.

□

Lemma 4.3.5 (Normal path decision)**.** *Assume w.l.o.g. that node A is the first node to decide. Let node A decide v in round r via the normal path decision. Then no node can decide $v' \neq v$ in round $r' \geq r$.*

Proof. We prove by contradiction. Assume ∃ node I that in round r' decides $v' \neq v$. We analyze separately the case when $r' = r$ and $r' > r$.

By the normal path decision condition, the best proposal $p_{Hi} \in \mathsf{Universal}_A$ is uniquely best in $\mathsf{Universal}_A$, is also uniquely best in $\mathsf{Existent}_A$, and $p_{Hi}.v = v$. If node I decides v' in round r, then $\exists p'_{Hi} \in \mathsf{Universal}_I$ that is uniquely best in $\mathsf{Universal}_I$ and $\mathsf{Existent}_I$. By Lemma 4.3.1, $\mathsf{Universal}_I \subseteq \mathsf{Existent}_A$ in round r, thus $p'_{Hi} \in \mathsf{Existent}_A$, and $\mathsf{Universal}_A \subseteq \mathsf{Existent}_I$ in round r, thus $p_{Hi} \in \mathsf{Existent}_I$. We are in the situation that $\{p'_{Hi}, p_{Hi}\} \subseteq \mathsf{Existent}_A$ with p_{Hi} being uniquely best, which means $p_{Hi}.r > p'_{Hi}$; at the same time, $\{p'_{Hi}, p_{Hi}\} \subseteq \mathsf{Existent}_I$ with p'_{Hi} being uniquely best, which means $p'_{Hi}.r > p_{Hi}$, which is a contradiction.

Assume ∃ node I that in round $r' > r$ decides $v' \neq v$. ∀ node I advancing to round $r+1$, node I proposes any best $p'_{Hi} \in \mathsf{Common}_I$ of round r. By Lemma 4.3.1, $\mathsf{Universal}_A \subseteq \mathsf{Common}_I \subseteq \mathsf{Existent}_A$ in round r. We now have that p_{Hi} is uniquely the best in both $\mathsf{Universal}_A$ and $\mathsf{Existent}_A$, but also p'_{Hi} best in Common_I. It must be that $p'_{Hi} = p_{Hi}$ and in particular $p'_{Hi}.v = p_{Hi}.v$ Because the sets are reset at the beginning of round $r+1$, and all proposals have value v, no node can ever decide any other value.

□

Lemma 4.3.6 (Agreement)**.** *No two nodes decide differently.*

Proof. The lemma follows directly from Lemma 4.3.5 and Lemma 4.3.4.

□

4.3.4 Liveness

Lemma 4.3.7 (Network schedule independent of priorities)**.** *The network delivery schedule at any node for the first and second* FSTSB *call in every round is independent of proposal priorities.*

Proof. Recall that the adversary is content-oblivious, i.e., it cannot observe node state or contents of messages on network links. Thus the adversary cannot link any proposal priority to its sender node before or during the first FSTSB call of each round r. Thus, message delivery and in particular the contents of Existent and B sets does not depend on proposal priority.

A similar property holds for the second FSTSB call: message delivery and in particular the contents of Common and $universale$ sets does not depend on proposal priority. □

Lemma 4.3.8 (Termination)**.** *Once a node decides, all nodes decide in the next round.*

Proof. Assume w.l.o.g. that the node is the first to decide, and decides v in round r. By Lemma 4.3.4 and Lemma 4.3.5, no node can decide $v' \neq v$ in round $r' \geq r$. Thus, all proposals that could be decided on in rounds $r' \geq r$ contain only value v. Once a node decided in round r, all nodes take the early decision path in round $r+1$, because all possible proposals have the same value v. Nodes decide v if they have not decided already, and all nodes exit. □

Lemma 4.3.9 (Decision probability)**.** *The probability P that each node decides in any round r is at least $t_b/n - p_t$, where p_t is the probability that two proposals tie for highest priority.*

Proof. Each node A may decide through the early path with probability P_e or the normal path with probability P_n. The probability to decide in a round is thus $P = P_e + P_n$ We compute the probability P_n that a node decides via the normal path in any round r. Because $P_e \geq 0$, we have $P \geq P_n$, thus P_n represents the lower bound for deciding in round r.

A node A decides via the normal path when the best proposal $p \in \mathsf{Universal}_A$ is uniquely best in both $\mathsf{Universal}_A$ and $\mathsf{Existent}_A$. We compute the probability P_b that the best overall proposal in the round is uniquely best and is in both $\mathsf{Universal}_A$ and $\mathsf{Existent}_A$. However, note that the normal path condition may also be true when p is not the best overall proposal in that round. Thus $P \geq P_b - p_t$.

Based on the FSTSB properties, each node A completing the first FSTSB call in any round has $|B_A| = t_b$. By Lemma 4.3.7, any proposal has the same probability t_b/n to be in B_A, thus also the highest priority proposal of the round $p_{Hi} \in B_A$ with probability t_b/n. The probability for *at least one node* to have $p_{Hi} \in B_A$ is the sum of the probabilities that each node has $p_{Hi} \in B_A$, i.e., the sum is t_b. Nodes that have $p_{Hi} \in B_A$ necessarily broadcast the proposal in the second FSTSB call. By Lemma 4.3.7, proposal $p_{Hi} \in \mathsf{Universal}_A$ with probability t_b/n.

Because $p_{Hi} \in \mathsf{Universal}_A \subseteq \mathsf{Existent}_A$ (Lemma 4.3.1), and p_{Hi} is the unique best proposal in that round, there cannot be a better proposal in any of these sets and the probability that node A decides via the normal path is $P_b = t_b/n$. As the final probability $P \geq P_b - p_t$, we conclude that $P \geq t_b/n - p_t$.

□

4.3.5 Asymptotic complexity

Lemma 4.3.10 (Time complexity)**.** *When $t_b = O(n)$, the expected termination of the* QSC *algorithm is $O(1)$ rounds.*

Proof. By Lemma 4.3.9, the first node to decide does so with probability $P \geq t_b/n - p_t$ in any round. Because $t_b = O(n)$, we expect the first node to decide in $O(1)$ rounds. By Lemma 4.3.8, all nodes decide and exit by the next round, thus the overall expected termination time is $O(1)$ rounds. □

Communication complexity. The communication complexity in QSC is mainly due to FSTSB. We show in the next section that FSTSB has a message complexity of $O(n^2)$ across the n nodes. The bit complexity of FSTSB is $O(n^3)$ if either $t_b = O(n)$ or $f = O(n)$, which is consistent with the largest value for f i.e., when $n = 2f + 1$. Together with Lemma 4.3.10, QSC has an asymptotic communication complexity of $O(n^3)$.

4.3.6 Optimizations

Round catch-up. Slow QSC nodes may lag behind quicker nodes by several rounds. It is feasible for a group of QSC-running nodes to complete a FSTSB step even if not all live nodes participate in the step, provided the group is big enough to meet the communication thresholds. Consequently, a group of nodes may be many rounds ahead of slower nodes.

Slow nodes may always execute *one by one* the steps that other FSTSB nodes have already completed, without compromising the safety of QSC. The key to guaranteeing safety despite slow nodes is to disregard a node's proposal if the node does not start a step before another node has already completed the step. Otherwise, the step-completing node might possibly miss proposals. Therefore, the only option for a slow node to catch up to step s is for it to adopt the sets of another node that has already completed step s. The TSB (and implicitly FSTSB) catch-up mechanism (Section 4.4.4) enables nodes to catch up one step by simply adopting the R and B sets of any other node. If a slow node catches up to the first FSTSB call in a QSC round, it learns Existent and B sets that satisfy FSTSB properties. If the node catches up to the second FSTSB call in a QSC round, it learns valid Common and Universal sets.

Enabling slow nodes to bypass many QSC rounds and catch up to quicker nodes is a further enhancement that has the potential to reduce decision latency on these nodes. There is one constraint to catching-up, however. A node that completes a QSC round must have all three sets Existent, Common, and Universal in order to decide, and choose and continue with a safe value. Thus, a node skipping multiple steps should only catch up to the first FSTSB call of a round. The node may then skip one additional step and catch up to the second FSTSB call. To implement the optimization, QSC may leverage FSTSB's multi-step catch-up (Section 4.4.4) and apply the round constraints above.

4.4 Threshold-Synchronous Broadcast (TSB)

This section describes TSB, a novel group broadcast abstraction that provides threshold-reliability regarding the messages broadcast and received by nodes. First, we describe the TSB properties and the group communication thresholds, and explain how to set the thresholds to obtain FSTSB. For context, we also compare TSB with related primitives, such as reliable broadcast and common core. Next we provide a construction for TSB based on message witnessing. We also give details on the "catch-up" mechanism that enables slow nodes to progress quickly through TSB steps and reach a threshold of faster nodes. Finally we argue that

the construction satisfies the TSB properties and we give the TSB communication complexity.

4.4.1 The TSB primitive

TSB operates on a group of n nodes communicating in the asynchronous network model and delivers to the caller nodes threshold-synchronous group communication. Up to f nodes can fail by crashing silently. Each node in the group that has not (yet) failed broadcasts one message to the others, and receives some of the messages broadcast by the nodes. The messages broadcast in a TSB instance - which we simply call TSB step - can only be received in the same TSB step. Once a node has completed a certain TSB step, it no longer accepts messages from that step. [1] If a node gets a message from a future step, it defers its processing until the node advances to that step.

We represent the threshold broadcast primitive as a function $\mathsf{TSB}(m, t_r, t_b, t_s) \rightarrow (R, B)$. When a node i calls $\mathsf{TSB}(m, t_r, t_b, t_s)$ at time-step s, the network broadcasts message m and returns two message sets R and B to the caller within one TSB step. R comprises at least t_r messages received by the node, each of which was broadcast by a node in that step (t_r specifies the threshold for receiving messages). B is a set of t_b messages, each broadcast by a node and received by t_s nodes during the same time step (t_b, t_s are the broadcast and spread thresholds, respectively). For the threshold-reliability properties to hold, all nodes participating in the same TSB step should employ the same thresholds However, nodes may return R and B sets that vary from those observed by other nodes.

Properties and non-guarantees. Formally, the properties that TSB provides to the caller node are as follows:

- Receive threshold: There is a node set $N_R \subseteq \{1, \ldots, n\}$ such that $|N_R| \geq t_r$, and R contains exactly the set of messages m_j broadcast by nodes $j \in N_R$ during step s.
- Broadcast threshold: There is a node set $N_B \subseteq \{1, \ldots, n\}$ such that $|N_B| \geq t_b$, and B contains exactly the set of messages m_j broadcast by nodes $j \in N_B$ during step s.
- Spread threshold: For every message $m' \in B$, there are at least t_s nodes whose message sets R to be returned from TSB in step s will include message m'. These node sets j that receive a particular message m' might be different for different messages $m' \in B_i$. Formally, $\forall$ node $i, \exists t_s$ nodes j s.t. $B_i \subseteq R_j$.

TSB makes no additional guarantees. TSB does not guarantee, for example, that node i's own message m is included in the sets R or B returned to node i. Further, two nodes i and j may get distinct sets $R_i \neq R_j$ and/or different broadcast sets $B_i \neq B_j$ returned from their respective TSB calls in the same step.

[1] This is readily accomplished by affixing a TSB step number to the messages; if a node receives a message with a lower TSB step, it simply ignores it.

One question is how failing nodes impact the properties of TSB. Due to the fact that TSB involves group communication, nodes that fail during a TSB step may affect the properties of other nodes. Clearly, if an excessive number of nodes fail, TSB may be unable to meet the t_r and t_b thresholds in the first two properties. It is more subtle, however, how a failing node may affect the t_s (spread threshold) guarantee. If a node fails after TSB has spread messages to it, the failed node will count towards the spread threshold for those messages. The intuition is, had the failing node completed the TSB step, the node would have received that message in its R set. Thus, the significance of the spread threshold is that, at the end of the step, TSB guarantees that messages in each node's B sets are spread to at least $t_s - f$ nodes that also finish the step.

4.4.2 Thresholds for FSTSB and comparison with other primitives

A particularly useful primitive, which QSC builds on, is *full-spread TSB* (FSTSB). Full spread means that, at the end of the step, all messages in B sets have been spread to all live nodes, in their corresponding R sets. In principle, we could get FSTSB by calling TSB(m, t_r, t_b, n). Note, however, that some nodes could fail prior to the beginning of the TSB step or before counting as spread witness for sufficient messages. In these cases, it is impossible to satisfy $t_s = n$ and the TSB step never completes. Note, however, that setting $t_s = n - f$ would not fulfill full spread either if fewer than f nodes fail.

Because we cannot specify full spread directly via t_s, we obtain FSTSB by relying on classical quorum intersection in the TSB implementation. Section 4.4.3 gives more details.

FSTSB versus related abstractions. A FSTSB primitive is missing from the classical distributed systems theory, and it is not easy to substitute it with alternative broadcast primitives that provide reliability guarantees.

One naïve attempt to emulate FSTSB may consist of launching n instances of *reliable broadcast*, one on each node. Reliable broadcast guarantees to the caller that its message is eventually received by all live nodes. However, reliable broadcast does not actually inform the caller when this desired event has happened or even when it can be guaranteed to happen. If each node just waits for the first t values, where $n - f \geq t > 0$, each node may end up with a set that shares no values with the sets of the other nodes. Indeed, this situation is possible even when $f = 1$: Node 1's message might be delivered last to node 2, after node 2 has already received t messages and moved on; node 2's message could be delivered last to node 3 and hence missed by node 3; and so on. By the time each node reaches its threshold t, it is missing one node's message - a different one for every node - and by the end of the protocol, no node's message has been received by every node. Thus, n instances of reliable broadcast fail to guarantee that even a single node's value is successfully received by all live nodes, which in FSTSB terms means that the B set is empty.

However, FSTSB does not in general offer perfect communication reliability, whereas n

instances of reliable broadcast eventually deliver to all live nodes all successfully broadcast messages. The disadvantage of reliable broadcast is that, without running binary consensus, nodes cannot pinpoint the moment in the execution when the delivery occurs. Running consensus is undesirable since it increases complexity.

A problem strictly easier than TSB is known as *common core* [16]. The common core is a set of $n-f$ values, and each node obtains a superset of the common core after broadcasting a value. However, the nodes cannot identify which values comprise the common core. In TSB terms, nodes executing the common core obtain R sets that overlap in $n-f$ nodes; nevertheless, each R set may contain more than $n-f$ messages. Nodes are unable to identify even one of the messages in the common core; in FSTSB terminology, nodes do not get explicit B sets.

Spreading *and identifying* a threshold of messages to all live nodes is the strength of TSB. The next sections presents a TSB construction.

4.4.3 Building TSB

To build FSTSB, we rely on classical quorum intersection. We proceed in two phases - spread and gather - ensuring that the quorums from both phases overlap. Algorithm 9 depicts the pseudocode. First, using tSpread, each node obtains its B set by spreading its message to t_s nodes using best-effort broadcast, and collecting t_b spread messages. Second, using tGather, each node collects $\geq t_r$ messages in its R set to ensure that $B_i \in R_j, \forall$ nodes i, j.

Algorithm 10 depicts the pseudocode for tSpread. We present the algorithm using the event-based notation of [32]. The first part of tSpread resembles an echo-broadcast algorithm, also used by other protocols [9, 69], but without the signatures as they are unnecessary in an environment with crash faults. Each node spreads its message through best-effort broadcast, relying on unicast echos from the nodes to understand when the message has spread to (or has been witnessed by) t_s nodes; once witnessed, the node best-effort broadcasts the message. The node also collects in its R set the messages it echoes. In parallel, the node collects t_b witnessed messages in its B set and finishes the algorithm once it does so, even if its own message is not yet witnessed.

Algorithm 11 depicts the pseudocode for tGather. Each node simply best-broadcasts its R set collected during tSpread, which contains the echoed messages. Each node gathers t_r messages sets and appends them to its R set. Messages in all B sets created during tGather each appear in the R sets of at least t_s nodes. Intuitively, when the t_r and the t_s quorums overlap, we ensure the FSTSB property that $B_i \subseteq R_j, \forall$ nodes i, j.

Lemma 4.4.1 (TSpread)**.** *If* $0 \leq t_s \leq n-f$ *and* $0 \leq t_b \leq n-f$*, Algorithm 10 implements* TSB$(0, t_b, t_s)$.

Proof. Let there be n nodes that that call tSpread. We first show that $\exists$ some node i that (1) completes tSpread and does not fail; and (2) $|B_i| = t_b$ and $\exists t_s$ nodes j s.t. $B_i \subseteq R_j$ that are either

Algorithm 9: TSB implements threshold-synchronous broadcast (node A).

Parameters : t_r, t_b, t_s the receive, broadcast, and spread thresholds, respectively
Function input : message m to broadcast
Function output: sets R and B of messages received
State : $R, B \leftarrow \{\}, \{\}$

upon call TSB-broadcast$m$ **do**
  $R', B \leftarrow$ tSpread(m) // Collectively get $\geq t_b$ messages witnessed by t_s nodes.
  $R'' \leftarrow$ tGather(R') // Gather the witnessed messages.
  $R \leftarrow R' \cup R''$ // The messages I witnessed directly via tSpread and indirectly via tGather.
  output TSB-deliver(R, B)

Algorithm 10: tSpread implements TSB$(0, t_b, t_s)$ (node A).

Parameters : t_s, t_b the spread and broadcast thresholds
Function input : message m to broadcast
Function output: sets R, B of messages
State : $R, B \leftarrow \{\}$
State : $echos \leftarrow 0$ // Counter for the number of echos for m.

upon call tSpread-broadcast(m) **do**
  best-effort broadcast [SEND, m] to all nodes j
upon *receiving a message* [SEND, m] *from node* j **do**
  $R \leftarrow R \cup \{m\}$
  send message [ECHO, m] to node j
upon *receiving a message* [ECHO, m] *from node* j **do**
  echos++ // By construction, this node receives echos only for its own message m.
  if $echos = t_s$ **then**
    best-effort broadcast [WIT, m] to all nodes j
upon *receiving a message* [WIT, m] *from node* j **do**
  $B \leftarrow B \cup \{m\}$
  if $|B| \geq t_b$ **then**
    best-effort broadcast [WIT, m] for all $m \in B$ to all nodes j // Ensure that all nodes eventually observe the exit condition.
    tSpread-deliver(R, B)

Algorithm 11: tGather implements TSB$(t_r, 0, 0)$ (node A).

Parameters : t_r the receive threshold
Function input : message m to broadcast
Function output: set R of messages received
State : $R \leftarrow \{\}$

upon call tGather-broadcast(m) **do**
  best-effort broadcast message [SEND, m] to all nodes j
upon *receiving a message* [SEND, m'] *from node* j **do**
  $R \leftarrow R \cup \{\langle j, m' \rangle\}$
  if $|R| \geq t_r$ **then**
    output tGather-deliver(R)

still in the step or completed it. Because this node i broadcasts its B set before completing the step, all nodes still in the step eventually observe the termination condition, which means they also complete the step and fulfill the tSpread properties.

Assume w.l.o.g. that no node has completed the step yet. Node i completes the step when

$|B_i| \geq t_b$, and it fulfills this condition after receiving t_b messages tagged with WIT. We show that at least t_b nodes send WIT messages during the step.

A node k that has not failed sends a messages m with tag WIT after receiving t_s ECHO messages for message m. Node k can only receive ECHO messages for its own message m that node k broadcast when calling tSpread-broadcast(.) Because messages are eventually delivered, at most f nodes fail, and no node existed the step, eventually at least $n-f$ nodes receive node k's message m, add it to their R sets, and ECHO it. Thus, unless node k fails, it eventually receives at least $n-f$ ECHO messages and, because $t_s \leq n-f$, node k eventually broadcasts a WIT-labelled message.

We need to show that $\exists t_b$ that broadcast each a WIT-labelled message during the step. First, there are t_b nodes that do not fail, because $t_b \leq n-f$. Additionally, by our assumption, no node completed the step, thus t_b nodes will broadcast WIT messages.

Node i eventually receives these WIT messages from the t_b nodes that do not fail, and adds them to its B set, Moreover, each of the WIT messages is in the R sets of t_s nodes. This completes the proof.

□

Lemma 4.4.2 (TGather)**.** *If* $0 \leq t_r \leq n-f$*, Algorithm 11 implements* $\mathsf{TSB}(t_r, 0, 0)$

Proof. Let there be n nodes that that call tGather. We show that a node i that does not fail completes the step and, since node i is chosen arbitrarily, any node i completes the step.

Node i completes the step when $|R_i| \geq t_r$, and it fulfills this condition after receiving t_r SEND messages. A node sends a SEND message when it starts the step. Because n nodes start the step, of which at most f fail, and $t_r \leq n-f$, there are t_r nodes that do not fail and broadcast a SEND message each, which eventually are delivered by node i.

□

Lemma 4.4.3 (Full-spread TSB (FSTSB))**.** *If* $f < t_s, t_r \leq n-f$, $1 \leq t_b \leq n-f$*, and* $t_r + t_s > n$*, Algorithm 9 implements* $\mathsf{FSTSB}(t_b)$.

Proof. Consider any nodes i, j that complete the algorithm without failing. Because node i does not fail, node i also completes the tSpread step. By Lemma 4.4.1, node i's set B contains only messages that node i knows to be in the R sets of at least t_s nodes. In other words, $\exists t_s$ nodes j that started the tSpread step (but might have not completed it) s.t. $B \subseteq R_j$. Consider any message $m \in B$.

All nodes that complete tSpread run tGather, where each node inputs its R set. By Lemma 4.4.2, every node receives R sets from t_r nodes.

Because $t_r + t_s > n$, and both $t_r, t_s > f$, there exists at least one node k that does not fail whose R set contains message m and that passes its R set on to every node that runs tGather. Because node j does not fail, node j also completes the tGather step and thus $m \in R_j$. Because this applies to all messages $m \in B_i$, we have that $B_i \subseteq R_j$. The reasoning above applies to all nodes i and j, thus the algorithm implements FSTSB.

□

Communication complexity. Considering messages of size $O(1)$, the TSB implementation above has $O(n^2)$ message complexity and $O(\max\{t_b, t_r\} * n^2)$ bit complexity across the n nodes. In tSpread, each node runs an echo-broadcast algorithm, and then a best-effort broadcast algorithm, for a total of $O(n^2)$ messages and $O(n^2)$ bit complexity. At the end, every node broadcasts its B set, for a total of $O(n^2)$ messages and $O(t_b * n^2)$ bit complexity. In tGather, each node runs a best-effort broadcast algorithm for set R, with a total message complexity of $O(n^2)$ and a bit complexity of $O(t_r * n^2)$.

For FSTSB, the message complexity across the n nodes is $O(n^2)$. The bit complexity depends on $t_r > f$ and t_b. If both f and t_b have size $O(1)$, then the bit complexity of the algorithm is $O(n^2)$. If either of them is $O(n)$, the bit complexity becomes $O(n^3)$.

4.4.4 Optimization: catch-up mechanism

One possible optimization is to enable nodes to skip through TSB steps by adopting another node's R and B sets. Note that this is possible because a node's sets do not contain any node-specific information. The only requirements is that any node's sets meet the group-wide properties and thresholds. Indeed, a threshold of nodes may advance quickly through TSB steps as soon as they meet the required thresholds, while other nodes may fall behind. To help slower nodes catch up, we could change the algorithm as follows. Every node starting a step broadcasts not only its message, but also its R and B sets of the previous step. A node in step s receiving a message from step $s+1$ immediately completes the step by adopting the R and B sets in that message, which fulfill the properties required of any node completing step s.

This optimization may lead to faster implementations in practice, while not changing the asymptotic communication complexity. A node broadcasting in tSpread its R and B sets together with its message leads to a total bit complexity of tSpread of $O((t_b + t_r) * n^2)$, and an unchanged message complexity of $O(n^2)$. Overall, the TSB communication complexity is $O((t_b + t_r) * n^2)$. Compared to the previous complexity of $O(\max\{t_b, t_r\} * n^2)$, both implementations amount to $O(n^3)$ when at least one of t_r or t_b is $O(n)$, or $O(n^2)$ when t_r and t_b are both $O(1)$

One remaining question is whether a node could catch-up multiple steps at a time. From TSB's standpoint, there is no obstacle: A node that skips from step s to step $s+k$, where $k > 1$, completes only step $s+k-1$ (by adopting the R and B sets of a node that completed step

$s+k-1$) and starts step $s+k$. The node does not complete steps $[s, s+k-2]$. Depending on the higher-level primitive that calls TSB, however, it may not make sense for nodes to arbitrarily skip steps. For example in QSC, nodes can skip any number of steps, as long as they do not skip to the middle of a QSC round. If desired, one could enforce that nodes skip at most one TSB step at a time. One could achieve that via suitable thresholds that require a majority of nodes in a step for any node to be able to complete the step, or by adding the assumption that messages are pairwise-ordered, e.g., via TCP.

4.5 Discussion

QSC **with different failure modes.** TSB and QSC seem to be suitable (with a few changes) for a Byzantine environment, as the earlier work of [52] indicates. It would be interesting to develop TSB and QSC for hybrid failure modes, that separates nodes that are silent forever - which affect liveness - from those that are Byzantine and eventually send a message - which affect safety. Specifically, consider that up to e (evil) nodes can Byzantine, and up to f nodes can crash (fail) or be silent forever. Denoting by V the set of Byzantine nodes, and by F the set of nodes that eventually crash, then $V \setminus F$ denotes those nodes that that are Byzantine and eventually send a message, and $F \setminus V$ denotes the honest nodes that crash. In essence, the set V is relevant for QSC safety, and the set F is relevant for liveness in TSB. This model is generic: When $e = 0$, then the model becomes the classical crash-stop model; when $f = 0$, then the model becomes the anytrust threat model. This separation might enable more efficient protocols for TSB and QSC.

4.6 Summary

We designed QSC, the first practical crash-fault-tolerant asynchronous consensus protocol that does not rely on common coins. QSC employs private node randomness that each node attaches to its proposal. Nodes disperse proposals using our novel group communication primitive, threshold synchronous broadcast (TSB), which provides step-synchronous message delivery atop an asynchronous network TSB guarantees that each node receives a common core of values and can identify a non-empty subset of values that are part of the common core. TSB also helps QSC be responsive by progressing at the speed of a threshold of nodes, and reduces design complexity via modularity. Although QSC does not break any asymptotic complexity bounds, QSC's design may enable Byzantine fault tolerant asynchronous consensus that does not require expensive distributed key generation protocols.

5 Conclusion and Future Work

We believe that distributed systems designers and practitioners can and should build reliable, responsive systems by making Lamport exposure and asynchrony a core consideration in their design. Our first key insight for limiting exposure is to place focus on user-observed guarantees. By recognizing that strongly-consistent globalized infrastructures often exhibit localized needs for access, such as users interacting through workplace collaborative applications, data confined to jurisdictions due to regulations, and patterns of economic activity, we proposed exposure-limiting systems that removing false or implicit dependencies unnecessary for the user's activity. We designed Limix, the first distributed metadata coordination service that enables global management in cloud environments while protecting local accesses from distant failures. Limix insulates global strongly-consistent data-plane services and objects from remote gray failures, network partitions, and slowdowns by ensuring that the definitive, strongly-consistent metadata for every object is always confined to the same region as the object itself. We also designed Nyle, a trust-but-verify distributed ledger architecture that limits transaction exposure in Byzantine environments. Both Limix and Nyle support several exposure metrics, such as jurisdictional boundaries suitable for regulatory reasons, or round-trip time that additionally bounds slowdowns. The autozoning scheme we propose ensures that any user can continue to access any strongly consistent object that matters to the user located at distance Δ away, while being insulated from failures outside a small multiple of Δ.

Our second key insight is that coordination protocols should ideally be responsive to adverse and changing network conditions, by progressing at the actual speed of the network. However the current ecosystem only infrequently employs asynchronous protocols, which are capable of managing such situations. We demonstrated though the design of QSC that it is possible to eliminate the common coin assumption and obtain a responsive algorithm in the crash-fault model. Although QSC does not break any asymptotic complexity bounds, it relies only on node-private randomness, which could make possible extensions in the Byzantine model that require only verifiable random functions instead of expensive distributed key generation protocols.

5.1 Future work

Dependencies across layered systems. One challenge is complex cross-layer dependencies. Today's web/cloud paradigm is addicted to considering the location of data, metadata, and code to be unimportant. While convenient for development, this paradigm increases Lamport exposure. While Alice and Bob collaboratively edit a document, for example, layering creates many dependencies beyond the document data itself: (1) Higher in the stack, web-based user interfaces may pull icons and JavaScript from all over the world. (2) Lower in the stack, Alice's requests crossing the public Internet depend on BGP routing and many network elements out of the application provider's control. (3) Network congestion or other loads on Alice's zone from requests coming from outside the zone might degrade Alice's access.

We might try to apply Limix across all layers in a system. This approach could in principle account for all dependencies, but would require many stakeholders to coordinate. A more tractable approach is to focus primarily on one layer, such as the application, accepting that uncontrolled residual Lamport exposure may persist in other layers. While likely impossible to eliminate, applications might reduce residual Lamport exposure from lower layers by building on reservation-based infrastructure proving performance isolation, such as MPLS tunnels or leased lines at the network layer.

Byzantine asynchronous consensus. With QSC, we show that it is possible to solve asynchronous consensus in the crash-fault model without needing common coins. In the crash-fault model, QSC removes the need for a shared secret set up by a dealer. It seems possible to adapt QSC to the Byzantine model, thereby removing the need for common coins. This could be a major step forward towards Byzantine asynchronous consensus protocols that require only verifiable random functions instead of expensive distributed key generation protocols.

Bibliography

[1] Technical details behind a 400 Gbps NTP amplification DDoS attack, 2015. https://blog.cloudflare.com/technical-details-behind-a-400gbpsntp-amplification-ddos-attack.

[2] Arbor security report, 2020. https://www.arbornetworks.com/resources/infrastructure-security-report.

[3] CoreOS etcd, 2020. https://coreos.com/etcd/.

[4] Local businesses are more trusted than large enterprises, finds survey by yell, retrieved 2020. https://smallbusiness.co.uk/local-businesses-trusted-large-enterprises-2538416/.

[5] CAIDA archipelago (ark) measurement infrastructure, retrieved 2020. https://www.caida.org/projects/ark/.

[6] Americans trust local governments over the federal government on covid-19, retrieved 2020. https://today.yougov.com/topics/politics/articles-reports/2020/04/27/americans-trust-local-governments.

[7] The Go pq driver, retrieved 2022. https://github.com/lib/pq.

[8] J. Abley and K. Lindqvist. Operation of anycast services, December 2006. RFC 4786.

[9] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, July 2019.

[10] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[11] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[12] Amazon. Amazon web services cloud regions. https://aws.amazon.com/about-aws/global-infrastructure/?p=ngi&loc=1, 2021.

[13] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. Sharding the shards: Managing datastore locality at scale with Akkio. In *Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

[14] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Large-scale

Network Attacks on Cryptocurrencies. *38th IEEE Symposium on Security and Privacy*, May 2017.

[15] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3):165–175, September 2003.

[16] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004. ISBN 0471453242.

[17] Baruch Awerbuch and David Peleg. Sparse partitions (extended abstract). In *In IEEE Symposium on Foundations of Computer Science*, 1990.

[18] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Principles of Distributed Computing (PODC)*, 1983.

[19] Blocknative. The ethereum merge: What it means for the network, retrieved 2022. https://www.blocknative.com/blog/ethereum-merge-proof-of-stake.

[20] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1), March 2003.

[21] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2), 1987.

[22] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[23] Cristina Băsescu and Bryan Ford. Immunizing systems from distant failures by limiting lamport exposure. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2021.

[24] Cristina Băsescu, Georgia Fragkouli, Enis Ceyhun Alp, Michael F. Nowlan, Jose M. Faleiro, Gaylor Bosson, Kelong Cong, Pierluca Borsò-Tan, Vero Estrada-Galiñanes, and Bryan Ford. Limiting Lamport Exposure to Distant Failures in Globally-Managed Distributed Systems. *arXiv*, 2022. https://doi.org/10.48550/arXiv.1405.0637.

[25] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[26] Christian Cachin and Jonathan A. Poritz. Secure Intrusion-tolerant Replication on the Internet. In *Dependable Systems and Networks (DSN)*, June 2002.

[27] Christian Cachin and Björn Tackmann. Asymmetric Distributed Trust. In *Conference on Principles of Distributed Systems (OPODIS)*, 2020.

[28] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Symposium on Reliable Distributed Systems (SRDS)*, 2005.

[29] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In *19th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[30] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology (CRYPTO)*, 2001.

[31] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[32] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[33] Ran Canetti and Tal Rabin. Fast Asynchronous Byzantine Agreement with Optimal Resilience. In *25th ACM Symposium on Theory of computing (STOC)*, pages 42–51, May 1993.

[34] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[35] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[36] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[37] Brad Chase and Ethan MacBrough. Analysis of the XRP Ledger Consensus Protocol. *arXiv*, 2018. https://doi.org/10.48550/arXiv.1802.07242.

[38] Xiaoqi Chen, Hyojoon Kim, Javed M. Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring tcp round-trip time in the data plane. SPIN '20, page 35–41, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380416. doi: 10.1145/3405669.3405823. URL https://doi.org/10.1145/3405669.3405823.

[39] David D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, August 1988.

[40] CoinDesk. What Is KYC and Why Does It Matter For Crypto?, retrieved 2022. https://www.coindesk.com/learn/what-is-kyc-and-why-does-it-matter-for-crypto/.

[41] McKinsey & Company. The 2020 McKinsey Global Payments Report, retrieved 2022. https://www.mckinsey.com/~/media/mckinsey/industries/financial%20services/our%20insights/accelerating%20winds%20of%20change%20in%20global%20payments/2020-mckinsey-global-payments-report-vf.pdf.

[42] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Computer Communication Review*, page 15–26, August 2004.

[43] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[44] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, SOSP '07, 2007.

[45] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), apr 1988.

[46] RFC Editor. Bgpsec protocol specification. rfc 8205, 2017. https://www.rfc-editor.org/rfc/rfc8205.html.

[47] Ethereum. Ethereum off-chain scaling., retrieved 2022. https://ethereum.org/en/developers/docs/scaling/#off-chain-scaling.

[48] P.T. Eugster, R. Guerraoui, S.B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *International Conference on Dependable Systems and Networks (DSN)*, 2001.

[49] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, March 2016. USENIX Association. URL https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-eyal.pdf.

[50] fastly. Fastly content delivery - lightning-fast delivery, retrieved 2022. https://fastly.com/.

[51] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[52] Bryan Ford. Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus. *arXiv*, 2019. https://doi.org/10.48550/arXiv.1907.07010.

[53] Bryan Ford, Philipp Jovanovic, and Ewa Syta. Que Sera Consensus: Simple Asynchronous Agreement with Private Coins and Threshold Logical Clocks. *arXiv*, 2020. https://doi.org/10.48550/arXiv.2003.02291.

[54] Georgia Fragkouli. Toward internet performance transparency. page 104, 2022. URL http://infoscience.epfl.ch/record/296205.

[55] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazières. Oasis: Anycast for any service. In *Proceedings of the Conference on Networked Systems Design & Implementation (NSDI)*, 2006.

[56] GeoDNS. GeoDNS. http://www.caraytech.com/geodns/.

[57] S. Gilbert and N. Lynch. Perspectives on the CAP theorem. *Computer*, 45(2), February 2012.

[58] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[59] Google. Google cloud regions. https://cloud.google.com/about/locations, 2021.

[60] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 169–184, 2016.

[61] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[62] Osama Haq, Mamoon Raja, and Fahad R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *International Conference on World Wide Web (WWW)*, 2017.

[63] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[64] Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. Mean-

ingful availability. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[65] Alex Hern. Fastly says single customer triggered bug behind mass internet outage. https://www.theguardian.com/technology/2021/jun/09/fastly-says-single-customer-triggered-bug-that-caused-mass-outage, May 2021.

[66] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[67] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[68] The Wall Street Journal. Amazon finds the cause of its AWS outage: A typo, 2017. https://www.wsj.com/articles/amazon-finds-the-cause-of-its-aws-outage-a-typo-1488490506.

[69] Philipp Jovanovic. ByzCoin: Securely Scaling Blockchains. Hacking, Distributed, August 2016.

[70] Sabrina Kall, Cristina Basescu, and Bryan Ford. Know-Thy-Neighbour: Approximate Proof-of-Location. 2019. https://www.epfl.ch/labs/dedis/wp-content/uploads/2020/01/report-2019-1-Sabrina-Kall.pdf.

[71] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Symposium on Foundations of Computer Science*, 2000.

[72] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards IP geolocation using delay and topology measurements. In *Internet Measurement Conference (IMC)*, October 2006.

[73] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

[74] Katharina Kohls and Claudia Diaz. VerLoc: Verifiable localization in decentralized systems. In *USENIX Security Symposium (USENIX Security)*, 2022.

[75] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[76] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *39th IEEE Symposium on Security and Privacy (SP)*, pages 19–34. IEEE, 2018.

[77] Cockroach Labs. CockroachDB: Ultra-resilient SQL for global business, 2020. https://www.cockroachlabs.com/.

[78] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16 (2), May 1998.

[79] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, December 2001.

[80] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo

Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[81] Tom Lianza and Chris Snook. Cloudflare outage. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/, Nov. 2020.

[82] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[83] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Symposium on Principles of Distributed Computing (PODC)*, 2020.

[84] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[85] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[86] David Mazieres. Paxos made practical. 08 2009. https://www.scs.stanford.edu/~dm/home/papers/paxos.pdf.

[87] Tony McGregor, Shane Alcock, and Daniel Karrenberg. The ripe ncc internet measurement data repository. In *Passive and Active Measurement (PAM)*, 2010.

[88] Microsoft. Microsoft azure cloud regions. https://azure.microsoft.com/en-us/global-infrastructure/geographies/, 2021.

[89] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Computer and Communications Security (CCS)*, 2016.

[90] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[91] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages. In *Principles of Distributed Computing (PODC)*, July 2014.

[92] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[93] Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. 2021.

[94] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[95] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.

[96] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. 2016. https://lightning.network/lightningnetwork-paper.pdf.

[97] PwC. PwC's legal services for FinTech, blockchain and digital assets, retrieved 2022. https://www.pwc.ch/en/services/legal/banking-fintech-blockchain.html.

[98] Michael O. Rabin. Randomized Byzantine generals. In *Symposium on Foundations of Computer Science (SFCS)*, November 1983.

[99] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[100] Ripple. Business impact, powered by crypto, retrieved 2022. https://ripple.com/.

[101] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[102] Salvatore Scellato, Cecilia Mascolo, Mirco Musolesi, and Jon Crowcroft. Track globally, deliver locally: Improving content delivery networks by tracking geographic social cascades. In *Proceedings of the International Conference on World Wide Web*, WWW, 2011.

[103] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[104] Jeff Seibert, Sheila Becker, Cristina Nita-Rotaru, and Radu State. Newton: Securing virtual coordinates by enforcing physical laws. *IEEE/ACM Transactions on Networking*, 22(3), 2014.

[105] Similarweb. Website traffic - check and analyze any website, retrieved 2022. https://www.similarweb.com/.

[106] Slack. Great teamwork starts with a digital hq, retrieved 2022. https://slack.com/.

[107] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *ACM Workshop on Hot Topics in Operating Systems (HotOS)*, 2021.

[108] Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001. doi: 10.1145/383059.383071. URL http://doi.acm.org/10.1145/383059.383071.

[109] Ahren Studer and Adrian Perrig. The Coremelt attack. In *ESORICS*, 2009.

[110] Bitcoin Suisse. The Travel Rule – Crypto Meets Global Regulation, retrieved 2022. https://www.bitcoinsuisse.com/research/outlook/2020/the-travel-rule-crypto-global-regulation.

[111] Min Suk Kang, Soo Bum Lee, and V.D. Gligor. The Crossfire attack. 2013.

[112] Yixin Sun, Maria Apostolaki, Henry Birge-Lee, Laurent Vanbever, Jennifer Rexford, Mung Chiang, and Prateek Mittal. Securing internet applications from routing attacks. *Communications of the ACM*, 64(6), may 2021.

[113] TCP. Transmission control protocol, September 1981. RFC 793.

[114] Pasindu Tennage, Antoine Desjardins, and Eleftherios Kokoris-Kogias. Mandator and Sporades: Robust Wide-Area Consensus with Efficient Request Dissemination. *arXiv*, 2022. https://doi.org/10.48550/arXiv.2209.06152.

[115] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *ACM Symposium on Theory of Computing*, pages 183–192, 2001. URL citeseer.ist.psu.edu/thorup01approximate.html.

[116] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-409-6. doi: http://doi.acm.org/10.1145/378580.378581.

[117] Stojan Trajanovski, Fernando A. Kuipers, Aleksandar Ilić, Jon Crowcroft, and Piet Van Mieghem. Finding critical regions and region-disjoint paths in a network. *IEEE/ACM Transactions on Networking*, 23(3), 2015.

[118] Travel Rule Information Sharing Alliance (TRISA). Decentralized Cryptocurrency Travel Rule Compliance, retrieved 2022. https://trisa.io/.

[119] Paul Francis Tsuchiya. The Landmark hierarchy: A new hierarchy for routing in very large networks. In *ACM SIGCOMM*, pages 35–42, August 1988.

[120] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[121] Vytautas Valancius, Bharath Ravi, Nick Feamster, and Alex C. Snoeren. Quantifying the benefits of joint content and network routing. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 41(1), 2013.

[122] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys*, 47(3), February 2015.

[123] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[124] Anshul Verma, Pradeepika Verma, Sanjay Kumar Dhurandher, and Isaac Woungang. *Opportunistic Networks. Fundamentals, Applications and Emerging Trends.* CRC Press, 2021. ISBN 9780367677305.

[125] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchain with asynchronous consensus zones. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[126] Patrick Wendell, Joe Wenjie Jiang, Michael J. Freedman, and Jennifer Rexford. DONAR: Decentralized server selection for cloud services. 2010.

[127] Bernard Wong and Emin Gün Sirer. Closestnode.com: an open access, scalable, shared geocast service for distributed systems. *SIGOPS Operating Systems Review*, 40, 2006.

[128] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

[129] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[130] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin

Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[131] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Principles of Distributed Computing (PODC)*, July 2019.

[132] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. Scion: Scalability, control, and isolation on next-generation networks. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[133] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

CRISTINA BĂSESCU

cristina.basescu@gmail.com

Cristina enjoys designing and building fast, reliable distributed systems. Her interests include fault tolerance and scalability in distributed systems, consensus protocols and network security.

RESEARCH EXPERIENCE

École Polytechnique Fédérale de Lausanne (EPFL) 01/2017 – 01/2023 (expected)

Ph.D. Candidate in the Decentralized and Distributed Systems Group

Thesis title: Building Systems Resilient to Failures, Partitions, and Slowdowns

Supervisor: Professor Bryan Ford

Description:

- Worked on the design, implementation, and evaluation of reliable and secure large-scale distributed systems.
- Led four research projects: Limix, its application to key-value stores and blockchains, and QSC. Limix is a framework that improves the reliability of a distributed system by reducing the system's exposure to remote failures and slowdowns, published at HotNets 2021. The design and evaluation of a Limix-based strongly-consistent key value store is available as a preprint. The challenges of a Limix-based blockchain appeared in a poster at IEEE Security & Privacy 2017 ("Papers" section). The follow-up project, Nyle, is under active development. QSC is an asynchronous Byzantine consensus protocol, under active development.
- Implemented Limix mostly in Go, along with Python and Java.
- Co-authored projects on efficient consensus, Sybil resistance through proof-of-personhood, and monetary policies for self-governing communities.
- Prepared teaching and practical sessions material for new graduate courses: Decentralized Systems Engineering and Information Security and Privacy.
- Teaching assistant for graduate courses ("Teaching" section).
- Co-advised the Master Thesis of Arnaud Pannatier, and also several semester projects for Bachelor and Master students.

Eidgenössische Technische Hochschule Zürich (ETH Zurich) 01/2013 – 02/2016

Research Assistant in the Network Security Group

Supervisor: Professor Adrian Perrig

Remarks: February 2013 as visiting researcher at Carnegie Mellon University.

Description:

- Worked on the design, implementation and evaluation of secure layer 3 network protocols. These protocols are scalable, handling hundreds of Gbps of traffic on Internet core routers.
- Led two research projects: SIBRA and Faultprints. SIBRA is a scalable bandwidth reservation architecture that prevents link flooding denial of service attacks, published at NDSS 2016. Faultprints is a protocol that localizes packet drop, delay, and modification at autonomous system level between a source and a destination, published at S&P (Oakland) 2016 ("Papers" section).
- Implemented some protocols and demos in the SCION code base (SCION is a new Internet architecture pioneered by Prof. Perrig's group). Implementations in Java, C, and Python.
- Co-authored protocols for on-the-fly key derivation, source authentication, path validation, and new Internet architectures. Results published at SIGCOMM 2014 and CCS 2014 ("Papers" section).
- Teaching assistant for graduate and undergraduate courses ("Teaching" section).
- Co-advised the Master Thesis of Lukas Limacher and the Bachelor Thesis of Dominik Roos.

PROFESSIONAL EXPERIENCE

Amazon Inc. Seattle 09/2012 – 11/2012

Software development engineer intern, Relational Databases (RDS) MySQL team

Project: Intercept failing rollback calls on MySQL databases made by AWS RDS customers, and apply patches that ensure a consistent database version after rollback.

IBM Research Zurich **06/2010 - 08/2010**

Research Intern, Storage Systems Group

Project: Intercloud Storage (ICStore), which investigates how to leverage multiple independent cloud storage providers to boost the privacy, integrity and reliability of stored data. I designed new fault-tolerant distributed algorithms, and implemented a skeleton code in Java.

Patent: Distributed, asynchronous and fault-tolerant storage system (US 20120323851 A1)

INRIA Rennes **03/2010 - 05/2010**

Research Intern, KerData Team

Project: Building a security layer for BlobSeer. BlobSeer is a distributed storage and versioning scheme for very large binary data objects. In the context of using BlobSeer as Infrastructure-as-a-Service, I developed a security layer to detect malicious clients, ranging from protocol breaches to denial of service, and take adequate actions.

Orange Romania **07/2008 - 10/2008**

Software development Intern

Project: Handling files with private content. I created a secure web service in JBoss.

JOURNAL AND CONFERENCE PAPERS

Limiting Lamport Exposure to Distant Failures in Globally-Managed Distributed Systems
Cristina Băsescu, Georgia Fragkouli, Enis Ceyhun Alp, Michael F. Nowlan, Jose M. Faleiro, Gaylor Bosson, Kelong Cong, Pierluca Borsò-Tan, Vero Estrada-Galiñanes, Bryan Ford
arXiv, https://doi.org/10.48550/arXiv.1405.0637, 2022

Baxos: Backing off for Robust and Efficient Consensus
Pasindu Tennage, Cristina Basescu, Eleftherios Kokoris Kogias, Ewa Syta, Philipp Jovanovic, Bryan Ford
arXiv, https://doi.org/10.48550/arXiv.2204.10934, 2022

Immunizing Systems from Distant Failures by Limiting Lamport Exposure
Cristina Băsescu, Bryan Ford
Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks (**HotNets**), 2021

Economic Principles of PoPCoin, a Democratic Time-based Cryptocurrency
Haoqian Zhang, Cristina Basescu, Bryan Ford
arXiv, arXiv:2011.01712v1, 2020

Poster: Low-Latency Blockchain Consensus
Cristina Basescu, Eleftherios Kokoris-Kogias, Bryan Ford
*Poster session, IEEE Security & Privacy (**S&P**), 2017*

On the Implementation of Path-Based Dynamic Pricing in Edge-Directed Routing
Jumpei Urakawa, Cristina Basescu, Kohei Sugiyama, Christos Pappas, Akira Yamada, Ayumu Kubota, Adrian Perrig
Asia-Pacific Conference on Communications (APCC), 2016

High-Speed Inter-domain Fault Localization
Cristina Basescu, Yue-Hsun Lin, Haoming Zhang, Adrian Perrig
*IEEE Security & Privacy **(S&P)**, 2016*

Scalable Internet Bandwidth Reservation Architecture
Cristina Basescu, Raphael M. Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, Jumpei Urakawa
*Network & Distributed System Security Symposium **(NDSS),** 2016*

Source-Based Path Selection: The Data Plane Perspective
Taeho Lee, Christos Pappas, Cristina Basescu, Jun Han, Torsten Hoefler, Adrian Perrig
International Conference on Future Internet, 2015

Mechanized Network Origin and Path Authenticity Proofs
Fuyuan Zhang, Limin Jia, Cristina Basescu, Tiffany Hyun-Jin Kim, Yih-Chun Hu, Adrian Perrig
*Proceedings of ACM SIGSAC Conference on Computer and Communications Security **(CCS)**, 2014*

Lightweight Source Authentication and Path Validation
Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, Adrian Perrig
*Proceedings of ACM Special Interest Group on Data Communication **(SIGCOMM)**, 2014*

Robust Data Sharing with Key-Value Stores
Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, Ido Zachevsky
Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2012

Brief Announcement: Robust Data Sharing with Key-Value Stores
Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Marko Vukolic
ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing ***(PODC)****,* 2011

Towards a Generic Security Framework for Cloud Data Management Environments
Alexandra Carpen-Amarie, Alexandru Costan, Catalin Leordeanu, Cristina Basescu, Gabriel Antoniu
International Journal of Distributed Systems and Technologies (IJDST), vol. 3(1), pag. 17-34, 2012

Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies
Cristina Basescu, Alexandra Carpen-Amarie, Catalin Leordeanu, Alexandru Costan, Gabriel Antoniu
International Conference on Advanced Information Networking and Applications (AINA), 2011

PATENTS

Distributed, asynchronous, and fault-tolerant storage system
Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Marko Vukolic
US Patent App. 13/956,682, 2012

AWARDS

Teaching assistant distinction EPFL Lausanne **Fall 2019**

Awarded for the Decentralized Systems Engineering course

Accepted at the MIT Media Lab Berlin Workshop, Blockchain Track **08/ 2018**

Designed and implemented a promise-keeping smart contract on Ethereum

Vrije Universiteit Amsterdam

Huygens Scholarship (prestigious program for excellent students) **09/2011 – 08/2012**
VU Fellowship **09/2010 – 08/2011**

IBM Research Zurich **2011**

IBM Invention Achievement Award for "Robust Data Sharing with Key-Value Stores" (paper reference in section Journal and Conference Papers)

University "Politehnica" of Bucharest

Merit Scholarship (top 1-5%) 7 semesters out of 8 **2005 - 2009**

First Prize at the Students Scientific Conference (Large Scale Distributed Systems section) **2009**
Project: Dimmunix Signature Validation

IBM Best Paper Prize at the Students Scientific Conference (Large Scale Distributed Systems section) **2009**
Project: Dimmunix Signature Validation

First Mention at Scientific Communications Session on Artificial Intelligence **2007**
Project: Analyzing and simulating the dynamics of a social network

TEACHING

EPFL Lausanne **01/2017 – present**

Teaching Assistant: Information Security and Privacy (graduate course, Spring 2017, Spring 2018, Spring 2019), Decentralized Systems Engineering (graduate course, Fall 2017, Fall 2019, Fall 2020, Fall 2021), Database Systems (graduate course, Spring 2019)

ETH Zurich **01/2013 – 02/2016**

Teaching Assistant: Informatics for Mathematicians and Physicists (undergraduate course, Fall 2013), Information Security (graduate course, Spring 2014), System Security (graduate course, Fall 2014, Fall 2015)

Vrije Universiteit Amsterdam **02/2012 - 03/2012**

Teaching Assistant: Distributed Algorithms (graduate course)

University "Politehnica" of Bucharest

Co-Designer: Operating Systems Course (undergraduate course) **01/2010 - 06/2010**
Responsible for writing/reviewing the seminars on virtual memory, threads synchronization, reviewing the assignments on Generic Monitor

Teaching Assistant: Programming Course (undergraduate course) **10/2008 - 02/2009**

EDUCATION

Vrije Universiteit Amsterdam **09/2010 – 08/2012**

M.Sc. in Parallel and Distributed Computer Systems (Research Master Program)

GPA: 8.9 / 10 (89%)

Master thesis: Fault Tolerance in ConPaaS (grade: 9 / 10)
Supervisor: Assistant Professor Guillaume Pierre
Description: ConPaaS is a runtime environment for hosting cloud applications, developed jointly by several universities and funded by two EU projects. I designed and implemented fault tolerance for ConPaaS, configurable according to the requirements of various ConPaaS applications. I designed several failure detection and recovery policies. Failure detection relies on either existent communication, or on new explicit messages. Failure recovery policies depend on where failed components can be restarted, e.g. on existent virtual machines (VMs) or on new ones. I decided that the design should allow any VM to detect and react to failures, therefore I also designed an algorithm to solve consensus between the VMs attempting to restart a failed component. Also, since ConPaaS applications may be stateful, I used a DHT to replicate the state on N other VMs, which allows to localize a VM storing some specific state, and to restart on that VM the failed component.

Selected Courses: Distributed Systems (Professor Maarten van Steen, grade 10/10), Distributed Algorithms (grade 9.5/10), Concurrency and Multithreading (grade 9.5/10), Cluster and Grid Computing (grade 9.5/10), Scientific Writing in English.

Selected projects: Implementation of a part of the project Collaborative Wikipedia Hosting. *Description*: The project is part of a PhD thesis written at the VU Amsterdam. Collaborative Wikipedia hosting replicates Wikipedia pages and stores them on untrusted peers. Users access pages through a front-end, which fetches page chunks from peers, validates their authenticity, and uses caching to ensure an upper bound on staleness. My task was to implement the front-end functionality, with considerable freedom for the technologies I chose. Some technologies initially seemed a good fit, but lacked versatility. Some had better performance measurements than others, depending on the test conditions, for instance the system load. There were ample opportunities for myself to question my choices, and then either justify or rethink my decisions. I implemented an Apache module for the web server functionality and to communicate with the collaborators. To store the metadata of pages cached on front ends, I used a Berkeley DB database. Moreover, since the wiki pages are stored on the collaborators as wiki text, I wrote a MediaWiki extension (MediaWiki is also used by Wikipedia) to combine all the templates contained in a wiki page, and to parse and render it.

University "Politehnica" of Bucharest **2005 - 2009**

B.Eng. in Computer Science, Faculty of Automatic Control and Computers

GPA: 9.84 / 10 (98.4%, **3rd of approx 300 graduates**)

Engineering Diploma Thesis: Dimmunix Signature Validation
Topic: deadlock immunity, static analysis, peer-to-peer cooperative system
Supervisors: Assistant Professor George Candea (EPFL) and Professor Valentin Cristea (UPB). Diploma thesis graded 10 / 10

Major: Computer Science and Engineering. Specialization: Systems of Base Programs (courses on Operating Systems Design, Compilers, Network Programming Systems, Databases)

Selected courses: Algorithm Analysis and Design, Parallel and Distributed Algorithms, Formal Languages and Automata, Operating Systems, Multiprocessor Structures, Microprocessor Based Systems Design, Computer Systems Architecture, Software Engineering, Programming Paradigms, Digital Computers, Computer Graphics, Computer Networks, Object Oriented Programming,

Assembly Programming

Selected projects: Building a Compiler, Syscall Intercept, Stateful Firewall, Generic Monitor, Simulating Virtual Memory, Analysis of BPEL engines, Text mining using WordNet

University "Politehnica" of Bucharest **2007**

Participant, GridInitiative Summer School. Introduction to grid computing, Installation of a Grid system (UNICORE)

National College "Gheorghe Roşca Codreanu" (Romania) **2001 – 2005**

High School Diploma
GPA: 9.81 / 10

Participant in National Contests (Country or District Stage) **2001 – 2005**

Physics, Computer Science, Mathematics

SKILLS AND COMPETENCES

Computer skills

Programming / Scripting Languages: Go, C, Java, Python, SQL, bash scripting
Software Tools / Frameworks: git, Kubernetes, Docker

Languages

Romanian (native)
English (level C2.1): TOEFL IBT, score 109 / 120 (2009), First Certificate Cambridge Examination in English (grade A)
French (level B1)
German (level A2)

Technical skills

Task-oriented thinking
Good focus on details
Analysis skills

Soft skills

Excellent presentation skills and technical writing
Good communication skills
Reliable
Team spirit
Leadership skills
Adaptable to new situations

MISCELLANEOUS

I am ambitious, persevering, and curious. I believe in hard work and continuous learning.
My hobbies include reading, skiing, traveling, hiking, going to the theater.