NASA/TM–20250005643

Distributed Systems Challenges in Wildland Firefighting Environments

Lawrence Dunn
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

July 2025

NASA STI Program Report Series

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at `http://www.sti.nasa.gov`

- Help desk contact information: `https://www.sti.nasa.gov/sti-contact-form/` and select the "General" help request type.

NASA/TM–20250005643

Distributed Systems Challenges in Wildland Firefighting Environments

Lawrence Dunn
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA

Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

July 2025

Acknowledgments

The first author conducted the work during summer internships in 2022 and 2023 at the NASA Langley Research Center in the Safety-Critical Avionics Systems Branch focusing on distributed computing issues arising in the Safety Demonstrator challenge in the NASA Aeronautics System Wide Safety (SWS) program. The authors would like to thank Kari Greer, Andreas Johansson, and Ron Kemnow, who provided the photographs in Figures 1, 2, and 3, respectively.

Available from:

NASA STI Program / Mail Stop 150
NASA Langley Research Center
Hampton, VA 23681-2199

Abstract

The System Wide Safety (SWS) program has been investigating how manned and unmanned aircraft can safely operate in shared airspace. Enforcing safety requirements for distributed agents requires situational awareness and coordination by passing messages over a communication network. Unfortunately, the operational environment will not admit reliable high-bandwidth communication between all agents, introducing theoretical and practical obstructions to global consistency that make it more difficult to maintain safety-related invariants. Taking disaster response scenarios, particularly wildfire suppression, as a motivating use case, this self-contained memo discusses some of the distributed systems challenges involved in system-wide safety through a pragmatic lens, offering many illustrations for clarity. We explain through figures and examples the essential concepts behind the continuous consistency model for shared memory proposed by Yu and Vahdat, arguing that this model has great relevance for safety-related distributed applications that run over adversarial and disruption-prone networks.

Contents

1 Introduction **3**
- 1.1 Continuity and Distributed Systems 3
- 1.2 Summaries of the sections 5

2 Coordination Challenges in Disaster Response **6**
- 2.1 Communication and User Safety 6
- 2.2 Communication Patterns in the Field 10
- 2.3 Data Collection and Processing 13

3 Introduction to Distributed Systems **16**
- 3.1 Physical Synchronization 18
- 3.2 Message Passing and Causality 18
- 3.3 Virtual Clocks 21
- 3.4 Message Ordering 29
- 3.5 Shared Memory 33
- 3.6 Semantics and Consistency 36
- 3.7 Strong Consistency Models 38
- 3.8 The CAP Theorem 42
- 3.9 Causal Consistency 46
- 3.10 Section Summary 47

4 Timestamped Anti-Entropy **49**
- 4.1 Assumptions 51
- 4.2 Message Log 51
- 4.3 Summary Vector 53
- 4.4 Acknowledgment Vector 64
- 4.5 Message Ordering 65
- 4.6 Message Purging 66
- 4.7 TSAE using Version Matrices 67

5 Continuous Consistency **68**
- 5.1 System Model 69
- 5.2 Write Propagation 72
- 5.3 Ideal States 73
- 5.4 Measuring Divergence 74
- 5.5 Numerical Consistency 77
- 5.6 Order Consistency 83
- 5.7 Real Time Staleness 89
- 5.8 Correctness Properties 90
- 5.9 Extensions 90

6 Conclusion **92**

Bibliography **93**

1 Introduction

Civil aviation has traditionally focused primarily on the efficient and safe transportation of people and goods via the airspace. Despite inherent risks, the application of sound engineering practices and conservative operating procedures has made flying the safest mode of transport today. Now, the industry's strong safety requirements make it difficult to integrate unmanned vehicles into the airspace, accommodate emerging applications, and keep pace with significant recent growth in commercial aviation. To that end, the Airspace Operations and Safety Program (AOSP) initiated the System Wide Safety (SWS) project to investigate technologies and methods that enable manned and unmanned aircraft to safely operate in shared airspace.

This memo surveys topics in computing that are relevant to maintaining system-wide safety across large, physically distributed data and communication systems. It is intended to be self-contained and accessible to a technical audience without a deep background in distributed systems. Our motivating use cases come from civil emergency response scenarios, especially wildfire suppression. This setting was chosen primarily for three reasons. First, improved technology for wildfire suppression is frequently cited as a national priority [30]. First responders often stress the need for new technology to improve communications and data sharing. Second, the rules for operating in the US national airspace are typically relaxed during natural disasters and relief efforts, so this a suitable environment for testing new technologies. Finally, this setting is an excellent microcosm for the general challenges faced by other, non-emergency applications.

1.1 Continuity and Distributed Systems

A central theme of this document is *continuity*, in the sense of continuous functions from mathematics. At a high level, a continuous function is one where small changes in the input lead to small, predictable changes in the output. A familiar example is the volume knob on a speaker: adjusting the knob slightly leads to a proportionate change in sound level, without abrupt spikes or drop-offs. Continuity underlies much of safe engineering. Many functions found in an engineering context are continuous, so small changes in a system's environment (e.g., temperature, pressure, wind speed) typically yield small changes in its behavior, at least for changes falling within a certain range. This predictability allows engineers to measure, model, and anticipate a system's performance with high confidence.

Software systems, by contrast, do not inherently exhibit continuity. For example, consider that flipping a single bit from 1 to 0 can mean the difference between a valid memory access and a segmentation fault, a positive number and a negative one, and a correct or broken authentication check. Unlike many physical systems, small changes in a program's runtime environment can produce large, discontinuous changes in its behavior, unless the system is designed to respond gracefully. For applications that model physical systems, continuity may arise naturally, but many desirable properties in a general computing context (e.g., strong consistency, defined in Section 3) are all-or-nothing: you either have them or you do not, without an obvious middle ground. This is challenging because, as we will show, this inflexibility often leads to situations where one desirable property must come at the expense of another, which must therefore be "sacrificed" by the system designer. Working in a more continuous framework can allow for a more balanced tradeoff between competing objectives, but it is not always clear what continuity means for software systems, nor how to achieve it.

The kinds of software applications under discussion here (examined in detail in Section 2) support emergency responders in the field. They comprise diverse computing components that exchange data across communication networks to provide what emergency workers call a *common operating picture* to users of the application. In computer science terms, these are *distributed systems*: systems composed of physically separated nodes that coordinate via message-passing over a network. What makes these systems "distributed" is that the timescale of sending messages across the network can be significant relative to other kinds of events, such as a change in environmental conditions that might threaten responders on the ground. Furthermore, the network is far from perfect, which means that delayed, out-of-order, or lost messages become central engineering concerns for application developers. Network performance may be difficult to control and predict—particularly in the emergency response environment, where systems face difficult operating conditions, such as damaged infrastructure. Naturally, all of this unreliable behavior in the message-exchange medium makes it challenging to build robust, predictable applications that rely on the network to coordinate.

In the distributed systems setting, developers must navigate delicate tradeoffs between competing objectives. One classic tradeoff, which is central to this document, stems from an inherent tension between building a system that responds to user requests quickly and one that prioritizes a strong level of global consistency between system components (the "common" aspect of the phrase "common operating picture"). This tension is brought to a head in the CAP theorem, defined in Section 3. CAP is a fundamental result ruling out an idealized system that maintains strong consistency and guarantees system availability despite a faulty network. A naïve but somewhat common framing of the CAP theorem is that it forces developers to "choose" between consistency and responsiveness, which is exactly the sort of harsh binary choice we seek to eliminate—safe engineering implies we want systems to satisfy multiple competing objectives, rather than preferring one idealized extreme over another. Therefore, the present document culminates in a discussion of one solution proposed in the literature (the *conit* model of Yu and Vahdat [42]) for escaping the dilemma presented by the CAP theorem. This model, one possible solution among many, offers the continuous flexibility required of a safe but efficient system that runs over an imperfect network.

Ultimately, a central thesis of this document is that system-wide safety in emergency communications is not just a matter of improving radios, routers, or bandwidth. Rather, it is also a problem of computer science: ensuring that the applications built on top of this infrastructure behave reliably *despite* the inherent challenges of their environments. These systems must be engineered to make highly efficient use of their limited resources while degrading gracefully in the face of harsh conditions. In short, these systems must react to their environment continuously, so that small failures or delays do not cascade into system-wide unpredictability.

1.2 Summaries of the sections

This document does not present any novel research. Rather, our contribution is to explain in accessible terms the key technical ideas within a single document to a technical but non-specialist audience, while also explaining the specifics of the use case to readers unfamiliar with scenarios envisioned for these applications.

Section 2 opens with a practical overview of disaster response and some of the computing challenges encountered in this setting. The heavily disruptive nature of the communications network in these environments raises issues fundamental to the science of distributed systems. Real-world examples from disaster response scenarios are presented that demonstrate how these challenges affect system-wide safety.

Section 3 summarizes fundamental concepts and mechanisms used in distributed systems, culminating in the classic CAP theorem for both the linearizable and sequential shared memory consistency models (Theorems 3.34 and 3.35). CAP is a "negative" result, meaning it proves that a distributed system cannot guarantee strong consistency and remain available to users when the communication network is disconnected. The practical implication is that agents in emergency response environments invariably have incomplete information about the global system. While the CAP theorem is often presented as an unfortunate prohibition, this section explains that CAP merely highlights a general tradeoff. Furthermore, real-world applications often exhibit a kind of "locality" that mitigates some of the constraints implied by the theorem, which developers should take into account.

Section 4 presents Golding's Timestamped Anti-Entropy (TSAE) protocol [15]. TSAE provides a fault-tolerant message propagation mechanism that ensures messages are eventually delivered to all parties despite an imperfect network. This implements a "weak" consistency model, so it is not directly subject to the CAP theorem. By itself, TSAE does not solve the fundamental problems contemplated by this document, but it provides a strong foundation upon which to build.

Section 5 describes a TSAE-based data replication mechanism suitable for applications that run over networks with frequent disruptions, where the application must exercise control over the consistency of replicated data. This framework is based on the theory of *conits* (short for "consistency unit") developed by Yu and Vahdat [42], which balances the competing objectives of consistency and availability in a quantifiable way. Using the conit framework allows applications to define units of replicated state of interest, enforce policies limiting inconsistency between their replicas, and adapt these policies dynamically in response to changing conditions. Because it provides fine-grained control without sacrificing the flexibility and resilience of weak consistency, this model is suitable when the tradeoffs implied by the CAP theorem need to be carefully calibrated to support both performance and safety.

We conclude in Section 6 by recapping the main themes in this document and highlighting areas for further investigation. Ultimately, building distributed systems requires design decisions tailored to the environment and application, and the specific techniques discussed in this document are suggestive examples, but not universal nor complete solutions. This leaves many opportunities to specialize the concepts discussed here to the real-world needs of users.

2 Coordination Challenges in Disaster Response

This section explores key aspects of disaster response, particularly firefighting, that shape the focus of this document. We highlight how real-world environments create fundamental challenges that require solutions based on distributed computing principles. Even with the best communications technologies, core issues arise when distributed agents need to coordinate their actions across wide areas.

Disaster response settings, like wildfire suppression or hurricane relief, are marked by systemic communications challenges. A 2023 report by the President's Council of Advisors on Science and Technology (PCAST) highlights the need to address "the vulnerabilities and shortfalls in wildland firefighter communications, connectivity, and technology interoperability" as its top recommendation for wildland firefighting modernization [30]. Many of these vulnerabilities and shortfalls stem from factors inherent to disaster response: remote locations, difficult terrain, damaged infrastructure, harsh weather, and limited power, to name a few.

Field agents often face high message loss, distorted signals, and unpredictable delays in communication. A cautious approach suggests preparing for the worst performance at critical times—conditions that demand urgent, reliable contact often coincide with network failures. Disasters often damage and degrade the communications infrastructure, which is accompanied by a sudden surge in user demand that can overwhelm a network completely. This was starkly evident in the immediate aftermath of the September 11^{th} attacks, when sudden user demand and severed trunk cables crippled New York public and private communication networks, including dedicated networks for first responders [32]. These failures later became the impetus for the creation of FirstNet [11, 16], a national public safety broadband network (NPSBN).

Unreliable networks make coordinating distributed agents a significant challenge. Coherent decision-making and coordinated action require consistency, meaning agreement on the data shared between agents. We define consistency more precisely in Section 3, but the concept is as follows: it is critical for everyone to agree which firetrucks should respond to which areas, where helicopters should land, which tasks should be prioritized, or which radio frequencies are in use. Achieving stronger standards for consistency requires sending more information in a shorter time frame, which places a heavier strain on the network. When a communications link is slow, system components may have to pause and wait before agreement can be reached, diminishing the efficacy of the system. To avoid waiting in such scenarios, standards for consistency may have to be relaxed, meaning distributed agents have less agreement, which comes with its own challenges. In summary, there is an inherent tension between global system consistency and a system that responds quickly to user requests—consistency takes time.

2.1 Communication and User Safety

We turn our attention to the implications of the consistency/responsiveness tradeoff from a user safety perspective. Operational safety depends on agents quickly gathering and responding to information about their environment. This information is relayed

Figure 1: A DC-10 airtanker, rated for 9,400 gallons, drops retardant above Greer, Arizona. Image credit: Kari Greer/US Forest Service.

through communication networks, so poor communication becomes a safety problem. When communication falters, agents face a difficult choice: either wait for more information before acting, or act now with incomplete knowledge. Both inaction and uninformed action carry risks. This dilemma is closely related to a fundamental computer science principle known as the safety/liveness tradeoff [2].

For example, consider the use of firefighting airtankers, particularly Very Large Airtankers (VLATs), which can carry over 8,000 gallons of water or fire retardant [37] (Figure 1). The largest VLATs can drop more than 20,000 gallons—about 170,000 pounds' worth—in a single pass. In the U.S., these drops are typically made from just 250 feet above the tree canopy [37], and sometimes lower in practice. This sort of maneuver can easily crush a ground vehicle [31]. In 2018, a firefighter was killed, and three others were injured, when an unexpectedly powerful drop from a Boeing 747-400 Supertanker knocked down an 87-foot Douglas Fir tree [7].

Improving firefighter communications can be expected to lead to better safety outcomes. One such improvement is through the use of applications like TAK, the Team Awareness Kit, developed by the U.S. military in 2010 and later released in a civilian version. Wildland firefighters are increasingly using TAK, extended with aftermarket plugins, on ordinary cell phones to coordinate their activities in the field (Figure 2). A key application of this tool could be tracking the real-time GPS

Figure 2: Screenshot of a firefighter using TAK, where the left panel shows a map and the right is a video stream from an air vehicle. Image credit: Andreas "AJ" Johansson

coordinates of firefighters for safety monitoring.

Given the risks of VLAT drops, a seemingly reasonable policy might be to disallow drops unless a VLAT's computers have up-to-date information about the location of ground personnel. Unfortunately, system-wide safety is not so easily achieved, as the proposed measure is precisely the sort of thing subject to the safety/liveness tradeoff. Before giving an example of this tradeoff, it is important to recognize a linguistic nuance: in the context of distributed systems, "safety" refers to a specific type of system property, which is not inherently related to the safety of people. A *safety* property is defined as a prohibition that stops a system from taking an action that is considered "bad." Here is a possible safety property for the example above:

> $\mathbf{P}_{\text{safe}}$: Ground agents are known to be at least 100 feet outside the drop zone, and this information is current to within 30 seconds, or airtankers will not perform a drop.

In the context of emergency response, most safety properties probably affect human safety in some way, but not all properties that affect human safety are "safety" properties in the technical sense. We do not have to look far to find an example: the next example is not a "safety" property, but it has implications for human safety.

In contrast to safety properties, a *liveness* property demands some kind of action from a system, usually one that is considered "good." A characteristic of liveness properties is that they place an upper bound on the allowable delay of something. A liveness property for our scenario might be the following:

> $\mathbf{P}_{\text{live}}$: A VLAT on the ground will take off and perform a drop within 20 minutes of receiving a request from the incident commander.

(The previous value comes from a PBS interview with the Chief of Flight Operations for Cal Fire, who cited 20 minutes as an upper bound on the response time for aerial firefighting units within designated areas of responsibility [39].)

Note that $\mathbf{P}_{\text{live}}$ is a liveness property, not a safety property in the narrow technical sense, but it impacts human safety: it might be critical for VLATs to perform drops quickly if a wildfire is threatening the safety of ground personnel. That is why the safety/liveness tradeoff is a problem: we will show there is a tradeoff between these concepts. Because human safety requires both "safety" and liveness properties, we cannot simply ignore the tradeoff. Throughout the remainder of this document, "safety" will always refer to the narrow meaning of the word, unless it is clear from context that the everyday definition is meant.

Safety and liveness are frequently dual mandates that cannot be guaranteed simultaneously. Such is the case in our example: though $\mathbf{P}_{\text{safe}}$ and $\mathbf{P}_{\text{live}}$ are both desirable, certain situations will force decision makers to prefer one over the other. Consider the fact that the wildland firefighting environment is frequently GPS-denied. Heavy smoke, multipath effects, and so on can easily prevent a consumer-grade cellphone from obtaining reliable GPS coordinates. Additionally, factors like a damaged radio tower or environmental obstructions like a tall mountain can prevent communications between the air and ground. Such conditions would prevent a VLAT's computers from knowing the locations of ground agents, which immediately presents a dilemma: should the crew proceed without knowing the locations of ground personnel, maintaining $\mathbf{P}_{\text{live}}$ at the cost of $\mathbf{P}_{\text{safe}}$, or should it be cautious and wait for more information, maintaining $\mathbf{P}_{\text{safe}}$ at the cost of $\mathbf{P}_{\text{live}}$? There is no simple answer, with either choice presenting a downside with respect to the broader goal of system-wide safety.

Besides the safety/liveness tradeoff, the previous example exhibits two other themes important in distributed systems, both of which will be explored further in this document. The first is the *epistemological* nature—concerned with what information is *known* by *whom*—of reasoning about distributed systems. This aspect is reflected in wording of $\mathbf{P}_{\text{safe}}$ in VLAT example: Ground agents are known (by the VLAT's computers) to be outside of a dangerous area. This situation requires a deeper and more sophisticated analysis than one simply considering what is true. Mathematically, the logic of distributed agents is not the ordinary propositional logic but the modal logic S5, which extends propositional logic with additional axioms governing knowledge. (The application of S5 to reason about distributed systems is the topic of [21], Chapter 8.) Distributing knowledge requires communication between agents over a period of time over the network, which is not instantaneous and reliable, and it is from these imperfections that the safety/liveness tradeoff arises.

The second aspect exhibited above, albeit negatively, is that of *continuity.* A continuous system can flexibly adapt to its environment, but a discontinuous system is rigid and may exhibit suddenly different behavior in response to only small changes in the environment, such as a transient network failure. The properties $\mathbf{P}_{\text{safe}}$ and $\mathbf{P}_{\text{live}}$ exhibit a stark lack of continuity because they are inflexible, all-or-nothing propositions. Suppose that agents are known to be 500 feet outside the drop zone, but the information is only current to within 31 seconds—this extra second technically violates $\mathbf{P}_{\text{safe}}$, though it should be inferable that the ground agents are well away from danger. In particular, this example highlights that system-wide safety is more of a quantitative concept than a Boolean (true-or-false) one. A distributed system in a network-challenged environment should exhibit smoothly varying properties in response to its inputs. Ideally one can "tune" the system's properties for the particulars of its environment at any moment. The technical aspects of this theme are the focus of Section 5.

2.2 Communication Patterns in the Field

We now consider some of the communication patterns that occur in wildland firefighting. Readers may be surprised to learn that the state of the art is somewhat primitive, largely due to the sparse permanent communications infrastructure that exists in this setting. This makes wildfires an interesting and generalizable example for other kinds of civil disaster environments where the network is unreliable.

One important concept to draw attention to is "geospatial locality of reference," so to speak, that system designers should consider. By this, we mean the concomitance of two observations which, while not guaranteed rules, are approximately true in many circumstances. The first observation states that nearby agents have aligned interests:

> Agents with the most urgent need to coordinate their actions will usually be located closer together and require similar kinds of information.

The second observation states that nearby agents have more reliable communications:

> Agents that are located closer together generally enjoy more reliable communications between them than agents that are far apart. Conversely, information that travels long distances tends to be delayed or degrade in quality.

These related observations are what is meant by simply the "locality" principle. Locality is a crucial factor to analyze because, as presented in Section 3, there are major theoretical and practical limits to how well agents can coordinate *globally*, meaning with all agents knowing and agreeing on everything. To the extent the system exhibits locality, coordination can be achieved using more efficient short-range communication than less efficient long-range communication. Here, "efficient" should be read broadly, measured with respect to things like battery life, message delay, reliability, cost-effectiveness, equipment weight, and so on. This raises the question of how to most efficiently utilize network resources to achieve adequate levels of consistency. Aspects of this question are revisited in Section 5, in the context of a framework for weighing the relative importance of updates.

2.2.1 Communication on the Ground

In the field, communication between firefighters and other agents is often facilitated by handheld (analog) land-mobile radio (LMR). These radios are inherently limited in their battery life, bandwidth, effective range, and ability to work around environmental factors like foliage and smoke.

As an alternative to using a radio, it is common for wildland firefighters in the field simply to shout commands and notifications to nearby personnel. This exhibits the locality principle: a substantial amount of communication occurs directly between nearby firefighters working on related tasks that can communicate without network infrastructure. In a future environment where agents might be equipped with body-worn sensors and or even some form of heads-up display (HUD), this sort of low-range local communication might be facilitated by relatively inexpensive, low-power technologies such as Bluetooth or mesh Wi-Fi, without the need for more sophisticated (and heavy) equipment.

Communication over a long distance requires infrastructural support, such as the use of cell towers and repeater stations. Typically, disaster response environments have scarce permanent infrastructure. In a wildland fire setting, one might find a small number repeaters mounted to a nearby watch tower, for instance, but not necessarily a cellular network. Ad-hoc infrastructure, such as Cells On Wheels (COWs) or Cells on Light Trucks (COLTs)—i.e., portable cellular towers—can sometimes be deployed on an as-needed basis if the location allows for it. Similar kinds of equipment can also be mounted to backpacks and carried into the field. A common issue is making sure that all equipment is properly configured, for instance that radios are listening on the correct frequencies. Configuration is especially critical when different agencies and groups need to interoperate—another problem highlighted during the September 11$^{\text{th}}$ attacks.

Use of centralized infrastructure comes with the potential for widespread failure when the infrastructure breaks down. For example, in California, the Ironside Mountain lookout/repeater station (seen in Figure 3) was destroyed during the 2021 Monument Fire, which burned approximately 223,124 acres over 88 days [8]. The Ironside Mountain station had strategic importance, being located on a tall ridge. According to a video blog from a volunteer firefighter involved in the incident, its loss prevented communication between operators on different sides of the ridge, in networking parlance creating a *partition* that lasted until crews could ascend the ridge to deploy a temporary station:

> "When [the Ironside Mountain lookout station] burned down the radio repeater went with it. And so communications were lost across the fire... one side of the fire couldn't talk to the other side.... So it was kind of a critical job to get that road cleared so that the radio crews could go back up there and set up a temporary radio tower." [18]

A scenario where communication between two groups is completely severed is exactly the sort of thing considered by the CAP theorem in Section 3.

Ground vehicles Large numbers of ground vehicles—sometimes on the order of 100 during a major response—are involved in wildfire suppression. Various firetrucks, bulldozers and similar vehicles are commonly used to control the landscape and perimeter of the fire. An advantage of vehicles is that they can carry heavier and higher-power communications equipment than a human. For instance, a vehicle could be equipped with a BGAN or VSAT satellite terminal to maintain a connection

Figure 3: The Ironside Mountain lookout and radio repeater station, shown here in September 2009, was destroyed by wildfire in 2021. Image credit: Ron Kemnow

back to a central location. Additionally equipping the vehicle with something like a Wi-Fi or cellular base station using the satellite connection as backhaul would let the vehicle act as a bridge between agents in the field and central coordinators such as incident commanders or 911 dispatchers.

2.2.2 Communication in the Air

Wildland firefighting increasingly involves the use of helicopters and fixed wing aircraft. Civil aviation has traditionally employed simpler communication patterns than this use case demands. For instance, aircraft equipped with Automatic Dependent Surveillance-Broadcast (ADS-B) monitor their location using GPS and periodically broadcast this information to air traffic controllers and nearby aircraft. This sort of scheme has worked well in traditional applications, where pilots typically only monitor the general locations of a few nearby aircraft. The locality principle is exhibited here, too: aircraft have the highest need to coordinate when they are physically close and therefore in range of each other's ADS-B broadcasts.

In our setting, a large number or aircraft, easily on the order of 10 or more, may need to operate in a small area, near complex terrain, during adverse conditions, often at low altitude. In other words, the demands are many and the margins for error are small. This sort of use case calls for more sophisticated coordination schemes between airborne and ground-based elements than solutions like ADS-B provide by themselves.

As aircraft generally have better line-of-sight to ground crews than ground crews have to each other, firefighters sometimes relay messages to air-based units over the radio, which in turn is relayed back down to other ground units. The locality principle comes into play for this sort of message relaying scheme, but in the negative direction: relaying allows knowledge to travel farther but requires more resources and effort, and the extended reach comes at the cost of introducing delays and possible degradation of message quality, as in the classic game of "telephone." Hence, this mode of communication has generally been reserved for more critical information.

The Communications Program of the Civil Air Patrol (a civilian auxiliary of the U.S. Air Force) is sometimes deployed to provide communications for firefighters on the ground using airplane-mounted radio repeaters. In the future, this sort of service could be provided autonomously by portable infrastructure mounted to unmanned aerial vehicles (UAVs), which might perform additional functions such as tracking the fire perimeter.

In future environments, we envision resilient networks formed from heterogeneous collections of smaller networks, incorporating various communication technologies such as digital radios, Wi-Fi, 4G/LTE, 5G, and satellite communications. Communications in the field may incorporate aspects of mesh networks and mobile ad-hoc networks (MANETs). Given the environmental challenges, we assume that two agents will often only have intermittent end-to-end connectivity, if any. Facilitating communication through such a dynamic and chaotic mobile network calls for a disruption-tolerant networking (DTN) architecture, which provides a custody transfer and store-carry-forward model that is resilient to disruption [33]. The exact form of such a network remains a question for future investigation.

2.3 Data Collection and Processing

Perhaps the most universally acknowledged expectation for future disaster response environments is a heavy reliance on data gathered from both humans and sensors. Besides improvements to communications that facilitate information sharing, we expect advances in machine intelligence to greatly influence how this data is handled.

Agents in disaster response environments will be both producers and consumers of data, and this data will need to be processed by humans and machines in ways that agents can readily make sense of to support their decision-making. We list some of the possible sources and types of pertinent data:

- Free-form communication, especially real-time or recorded voice messages broadcast to many agents at once, which may need to be processed by machines to extract the most pertinent information into a more actionable format

- The exact or estimated location of victims, firefighters, vehicles, hazards, etc. displayed on applications like TAK
- Medical information gathered from victims, perhaps stored in and collected from electronic triage tags [24]
- Data about current and predicted fire behavior gathered from systems like the Fire Integrated Real-time Intelligence System (FIRIS) or NASA's Fire Information for Resource Management System (FIRMS)
- Weather data from the National Weather Service
- Topographic information about the terrain, highlighting for instance the location of rivers and roads that could form a fire control line
- Planned escape routes, rendezvous points, safety zones, and landing zones
- Availability and dispatching of assets, e.g. ambulances, airtankers, or crews on standby, such as the prototype application considered by Monares et al. [28]

In a perfect environment, such information would be shared with all necessary agents in whole and instantly. In reality, agents will be presented with information that is sometimes incomplete, out of date, or contradictory—for example, two 9-1-1 callers might give conflicting details, or two first responders may give conflicting reports about whether a structure has been evacuated already. These sorts of problems are further exacerbated by an unreliable network, where messages can be dropped, rearranged, or delayed. A competing concern is that the information presented will be *overcomplete*, filled with petty details that distract agents from their critical tasks.

In some ways, future systems for disaster response will bear resemblance to future systems for war fighting, such as the conceptual *Internet of Battle Things* (IoBT) [19]. To quote from that paper, agents "under extreme cognitive and physical stress" will be subject to a highly dynamic and dangerous environment. Various kinds of technology will assist humans by providing data to support sensemaking, but a contraindicating concern will be flooding agents with a "massive, complex, confusing, and potentially deceptive ocean of information." To avoid "swimming in sensors and drowning in data" [23]:

> "Humans seek well-formed, reasonably-sized, essential information that is highly relevant to their cognitive needs, such as effective indications and warnings that pertain to their current situation and mission." [19]

The field of Human-Computer Interaction (HCI) is concerned with "design, evaluation and implementation of user interfaces for computer systems that are receptive to the user's needs and habits." [6] As emergency control becomes more data-driven, particularly in hubs like dispatch centers that aggregate diverse streams of information, the challenge of ensuring users can interact effectively with these systems will become increasingly important. We propose that researchers in HCI take up the question of how agents under stress can process and respond to the flood of complex information they may face. Relevant topics for exploration include structuring interfaces to avoid cognitive overload and facilitate intuitive control. Attention should also be given to helping agents avoid subtle but critical mistakes, such as dispatching resources to the wrong location—for instance, *S. Example Rd.* instead of *N. Example Rd.*

2.3.1 Adversarial Behavior

One feature of the Internet of Battle Things worth highlighting is "the adversarial nature of the environment." This feature is common also to disaster environments, whether resulting incidentally from "fog of war" effects or deliberately caused by malicious actors seeking to exploit a civil disaster. Section 2.2 cited a real-world example of a critical communications station destroyed by wildfire, perhaps comparable to an attack by enemy forces.

There is a growing trend in disaster response to rely on "crowdsourced" information, where public safety officials process reports from the general public over non-traditional channels like social media. However, a significant vulnerability of crowdsourcing information is the potential for confusing of contradictory reports, which can resemble intentional deception. Rumors frequently plague disaster relief environments, which are quite susceptible to misinformation. For instance, during Hurricane Harvey in 2017, there were unconfirmed rumors of shots being fired at volunteer rescuers [38]. Tracing reports back to their source is often difficult. Even fully malicious activity like "swatting"—placing a fake 911 call to cause a large police response—is often observed in public safety. Whether misinformation is spread with malicious intent or through well-meaning confusion, the proliferation of false information in this chaotic environment can have adversarial effects. This should be anticipated as part of a careful approach to modernizing systems in this space.

2.3.2 Allocation of Network Resources

Communications in disaster response environments might even be less reliable than in the battlefield (setting aside offensive behavior like signal jamming), requiring a greater emphasis on the preservation of scarce network resources. For instance, a group of volunteer firefighters would have fewer resources than a tactical military unit, relying on commercial off-the-shelf (COTS) equipment rather than best in class hardware like sophisticated handheld satellite links. High bandwidth channels will often be in short supply, while adverse conditions like inclement weather or dense smoke may as well be assumed.

Given the heavy reliance on data and the scarcity of reliable communication channels, we expect a complex interaction between the high-level needs of distributed applications (e.g. an application for sharing real time weather data) and low-level concerns about network resources. This is because only the applications have enough information to determine which data is the most important and must be shared with whom first, which has ramifications for network-level mechanisms designed to prioritize important messages.

There is a widely accepted wisdom in computing—the end-to-end principle [34]—which suggests that applications should not make assumptions about the network, and that the network should be relatively agnostic to high-level application logic. However, in natural disaster environments where resources are scarce and reliable communication is critical, these subsystems may need to be more tightly integrated in how they influence each other to achieve the best performance. This approach would not contravene the end-to-end principle, but would involve carefully considering its application in this relatively extreme context.

Consider a centralized data fusion application running in an edge data center.[1] This application could detect critical events like a fire crossing a control line (a phenomenon called *slopover*) and alert ground responders. It might also warn responders who have strayed too far from an escape route or safety zone. These are high-priority notifications, so it would be worthwhile to allocate scarce network resources to convey them to the relevant parties in real-time.

On the other hand, while it may be beneficial for each firefighter to have real-time information about the location of every other firefighter, this may not always be critical. If transmitting this data strains the network, then perhaps only the general location of nearby teams should be sent. If the network is extremely constrained, communication may be restricted to only information strictly relevant to preserving life to ensure swift, reliable delivery. Thus, network allocation is a dynamic calculation influenced both by the criticality of the information (which is determined by the application logic) and the availability of network resources at a particular location. Network services should provide mechanisms like Quality of Service (QoS) indicators to allow prioritizing certain communication. Such mechanisms can be incorporated into a control loop where applications generate feedback that drives the decision-making process in lower-level parts of the network. However, simple QoS mechanisms may not be enough—even the routing protocol of the network may need to be more specialized to higher-level applications than in traditional environments.

3 Introduction to Distributed Systems

In this section, we distill two core topics in the theory of distributed systems: causality and timekeeping (two closely related ideas), and shared memory consistency. Our discussion is primarily informed by the manuscripts of Coulouris et al. [9] and Kshemkalyani and Singhal [21]. We focus on building applications relevant to the scenarios described in Section 2, aiming to highlight obstacles and strategies for developing distributed systems that can endure the delays and disruptions inherent to these communication-challenged environments. Readers interested in a summary of the highlights may skip to Section 3.10.

[1]An *edge* data center is one located closer to a network's edge, nearer to users, to provide low-latency communications for time-sensitive applications. Edge centers support applications that require significant amounts of information processing—enough that the application must be hosted in a datacenter, where compute resources can be scaled dynamically, rather than colocated with users where resources are limited.

At its core, a distributed system is a network of independent entities working together to solve problems too complex for any one part to tackle alone. From a bird's-eye view, the systems we envision are intricate and complex, made up of diverse, interconnected elements: field agents like firefighters, their handheld devices, airborne and ground vehicles loaded with communication and computing tools, swarms of sensors and Internet of Things (IoT) devices, and so on. These decentralized components operate alongside more centralized hubs: data fusion centers, incident command (IC) posts, public safety answering points (PSAPs), and emergency operations centers (EOCs). We imagine these components being woven together by a patchwork of communication technologies ranging from analog and digital radios to Bluetooth, Wi-Fi, LTE, 5G, satellite communications, and ad-hoc mesh networks like Meshtastic[2], DECT-2020 NR[3], and others. The system is thus a dynamic mosaic of elements cooperating to protect lives and property.

Given the unpredictable nature of the environment and the locality principle outlined in Section 2, communication between edge components—such as field operators—and centralized hubs is often inconsistent, sometimes available only intermittently. As a result, information flow is subject to appreciable delays compared to the timescale of critical events like a fire shifting direction or a dangerous condition being detected. In other words, the computing landscape is unmistakably *distributed*. Singhal and Shivaratri [35] define a distributed computing system as:

> "A collection of computers that do not share common memory or a common physical clock, that communicate by message passing over a communication network, and where each computer has its own memory and runs its own operating system."

This stands in contrast to a centralized computing environment, where processes can seamlessly share data through common memory, and memory access times are considered negligible.

For our use cases, message-passing latencies are not only significant but unpredictable and difficult to control. As a result, we can assume that some parts of the wider system will not have instantaneous, complete knowledge of every new piece of information. Only a few components, if any, will be able to maintain a global systemwide awareness. While deploying additional infrastructure in the field, such as Cells on Wheels (COWs), can help, the inherently distributed nature of the environment cannot be fully overcome or abstracted away. This reality must be embedded in the design of the software and networking architecture itself. Typically, this manifests in a shared "middleware" layer to coordinate the moving parts, ensuring they function as a unified system.

The fragmented flow of information presents several challenges for system designers. One of the foundational difficulties is that unpredictable latencies make it difficult for components to maintain a common understanding about the global sequence of events. For similar reasons, it becomes challenging for processes to

[2]The website of the Meshtastic project describes it as an "open source, off-grid, decentralized, mesh network built to run on affordable, low-power devices". See `https://meshtastic.org`.

[3]DECT-2020 NR is a non-cellular 5G standard intended for Internet of Things (IoT) operations. For a technical discussion see [20].

synchronize and agree on shared values, such as the current number of firetrucks available for dispatch. The remainder of this section will delve into these issues with more technical depth.

3.1 Physical Synchronization

A lot of challenges in distributed computing could be straightforwardly overcome if we assume that all participants have instantaneous access to a common time base, i.e. synchronized clocks. Let us explore why fine-grained synchronization is not a tenable assumption for all purposes.

Physical clocks, especially consumer-grade ones, suffer from *drift*, which is to say they do not all run at the same rate. Experienced IT administrators will testify that clocks can also be prone to misconfiguration. An incorrect date, time, timezone, or daylight saving time policy setting is a common source of IT issues, typically causing time-based security mechanisms like Transport Layer Security (TLS) authentication to misbehave. Consider also that devices may spend a long time sitting unpowered in storage without maintaining an always-on clock. For these sorts of reasons, we would not want to rest the integrity of a safety-related system on the assumption that a numerous and diverse assortment of devices have precisely synchronized internal clocks.

Clock drift can be corrected for using, for instance, signals from GPS satellites, but as mentioned in Section 2, civil disaster environments are frequently GPS-denied: factors like mountainous terrain, heavy smoke, and subterranean operations can lead to errors or block signals entirely. Protocols like the Network Time Protocol (NTP) [26] work to bring clocks into synchronization with respect to an authoritative source. On the public internet, NTP typically achieves synchronization to within values on the order of tens of milliseconds [27], but it is not clear the level of synchronization that can be expected from NTP in the sorts of use cases we have in mind. A field device initialized without internet access may have no idea what the time is, but it must still operate.

For distributed systems, the critical feature of time is that *the future cannot influence the past* [25]. Fortunately, this property can be enforced using mechanisms that do not rely on measuring real time. Below, we explain how "logical" clocks can be used to measure and enforce a key relation between events, this being their *causal precedence*.

3.2 Message Passing and Causality

We model a distributed system abstractly as a fixed set $\mathcal{P} = \{P_1, P_2, \ldots P_N\}$ of N *processes* which undergo atomic (indivisible) state changes known as *events*. Events are divided into three types: internal events, representing state changes inside a single process, and send and receive events corresponding to *messages* passed between processes. Note that this framework is quite abstract and applies to any kind of packet-based communication technology. To draw out the core issues surrounding messaging, the diagrams in this section do not depict any internal events, as they represent state changes that are not directly viewable to the network or other processes.

Throughout the section, processes and networks are opaque blackboxes, which concentrates our attention on the ramifications of unpredictable network latencies. We implicitly assume the reliable asynchronous network model: when a message is sent between processes, it certainly arrives at some point in the future, but we cannot say anything about when or in what order compared to other messages. At times, we consider the possibility that a message may never arrive. The choice depends on which networking technology (or which layer of the OSI networking model [10]) is under consideration.

Figure 4 illustrates a series of time diagrams for messages exchanged between three processes: P_1, P_2, and P_3. The x-axis represents the flow of real time from left to right, which each process represented by a worldline depicting the events occurring within that process. Each message, m, originates from a send event m^{send}, marking the moment the message is dispatched across the network by its source process. The delivery of the message corresponds to a receive event, m^{recv}. For now we assume messages have a single receiver. We write subscripts on messages to distinguish them for clarity, but these are not inherent to the messages themselves.

In these diagrams, arrows connect corresponding send and receive events, with their diagonal slant representing the latencies experienced as messages traverse the network. Because messages arrive with varying delays, they might arrive in a different order than they were sent in. In Figure 4a, for example, P_1 sends messages m_2 and m_4 sequentially, but m_4 arrives before m_2, which might occur if P_1 and P_3 are separated by a high-latency communication link. In Figure 4b, m_1 is the first message sent but the last to be delivered, potentially indicating a deteriorating link between P_1 and P_3, perhaps due to increased distance or inclement weather.

For many applications, it is critical to maintain a natural ordering of events known as *causal precedence*, or Lamport's "happens before" relation [22]. To formalize this, we first consider the intuitive way to order events within a *single* process:

Definition 3.1. For two events e and e' occurring in process P, we write $e <_P e'$ if e occurs before e' in P's worldline.

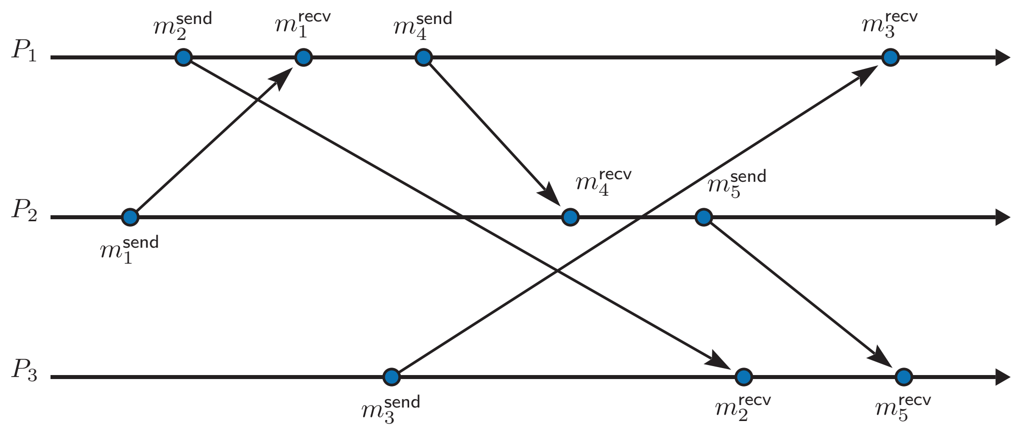
The previous definition is local to one process and unambiguous, as we assume events within a process occur at discrete, non-overlapping points in time. To extend this to a system-wide definition of causal precedence, we relate corresponding send and receive events, then take the transitive closure of the relation.

Definition 3.2 (Causal precedence)**.** We define a binary relation $\to$ on the set of events as follows:

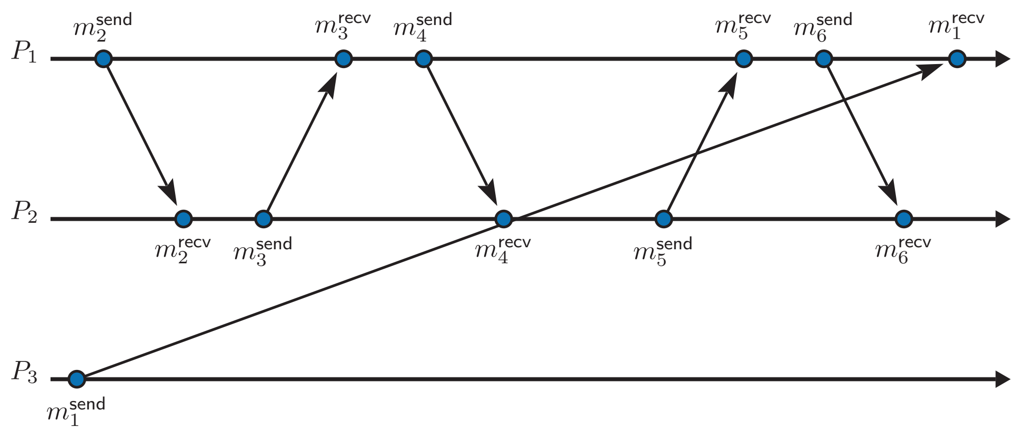
$$e \to e' \iff \begin{cases} e <_P e' \text{ for some process } P \textbf{ or} \\ e = m^{\mathsf{send}} \text{ and } e' = m^{\mathsf{recv}} \textbf{ or} \\ \text{there is some } e'' \text{ such that } e \to e'' \text{ and } e'' \to e' \end{cases}$$

If $e \to e'$, we say e has *causal precedence over* e' or *happens before* e'.

Visually, $e \to e'$ holds when one can put a finger on e in the diagram and trace a "path of causality" to e' by following worldlines or arrows. We use the notation $e \not\to e'$ to mean $e \to e'$ does not hold. Note that $e \not\to e'$ does not imply $e' \to e$.



(a) P_1 has a somewhat lower-latency connection to P_2 than to P_3



(b) P_1 has a much lower-latency connection to P_2 than to P_3

Figure 4: Message-passing time diagram examples

Incidentally, "causal precedence" and "happens before" can be misnomers, as $e \to e'$ only conveys the possibility that information from e could have influenced e'. The requirement to "not let the future affect the past" means that if e might have influenced e', then applications must avoid situations where, from the user's point of view, it appears that e' happened before e. For example, this proscription means an application cannot let an "answer" appear before the underlying "question".

Example 3.3. Figure 5 illustrates the causal precedence relation corresponding to the time diagrams in Figure 4. For readability we suppress redundant transitive arrows. The visual difference between Figures 4b and 5b reflects the fact that causal order only captures a logical relationship between events, but does not reflect their absolute time or within which process they occurred.

Mathematically, causal precedence is an irreflexive partial order: *irreflexive* because $e \not\to e$ (an event does not precede itself), and *partial* because any two events e and e' may satisfy neither $e \to e'$ nor $e' \to e$.

Definition 3.4 (Logical synchronicity)**.** Events e and e' that are not related by causality are said to be *logically synchronous*, denoted $e||e'$.

Note that logical synchronicity is not usually transitive, meaning it is possible to have $e||e'$ and $e'||e''$ but not $e||e''$. Relations like $(||)$ that are reflexive and symmetric but not necessarily transitive are sometimes called *compatibility relations*.

Example 3.5. In Figure 5a, $m_1^{\mathsf{recv}}||m_2^{\mathsf{recv}}$ and $m_2^{\mathsf{recv}}||m_4^{\mathsf{send}}$, but $m_1^{\mathsf{recv}} \to m_4^{\mathsf{send}}$. In Figure 5b, m_1^{send} is logically synchronous with every event except m_1^{recv}, but those other events are totally ordered by causality and not synchronous with each other.

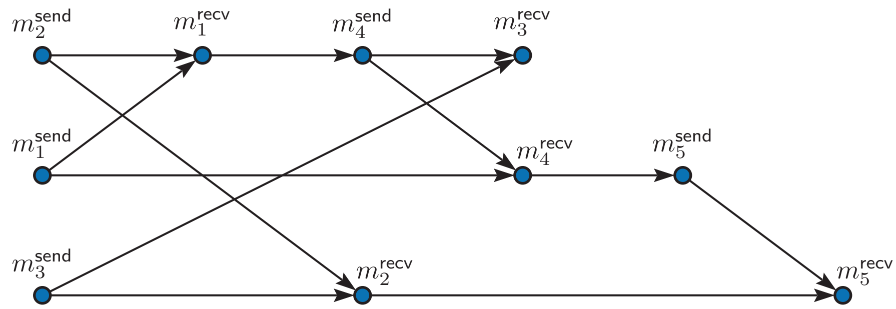
3.3 Virtual Clocks

Distributed applications systematically track causality by employing *logical* clocks, which measure the logical flow of time by timestamping events with (possibly sets of) non-negative integers that are advanced according to certain rules. The three major variants are scalar, vector, and matrix clocks, which form a kind of spectrum. Scalar clocks are simple but provide coarse-grained information, while vector and matrix clocks track increasingly more precise information at the cost of greater administrative overheads.

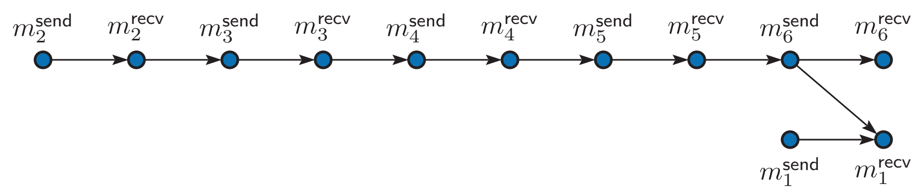
All processes timestamp their events using their local clocks. For each event e, let $C(e)$ denote the timestamp attached to that event. The fundamental property we want to satisfy is that if e causally precedes e', it should receive a lesser timestamp. This is called the clock consistency condition, or simply the clock condition.

Definition 3.6. A system of timestamps satisfies the *clock consistency condition* if the following monotonicity property holds:

$$\text{For all } e \text{ and } e', \text{ if } e \to e' \text{ then } C(e) < C(e') \tag{CC}$$



(a) Causal precedence among the events in Figure 4a



(b) Causal precedence among the events in Figure 4b

Figure 5: Causal precedence relations for Figure 4 (transitive arrows not shown)

This notation states that if one event causally precedes another, then the earlier one receives a lesser timestamp. Somewhat subtly, the clock condition does *not* imply that we can decide if events are causally related by comparing timestamps. Rather, it provides a way of *ruling out* causal precedence. This is seen by expressing (CC) in terms of the following logically equivalent condition.

$$\text{For all } e \text{ and } e', \text{ if } C(e') \le C(e) \text{ then } e \not\to e' \qquad \text{(CC}'\text{)}$$

If (CC′) holds, we can be sure that a particular sequence of events $e_1, e_2, e_3 \ldots$ does *not* list any e before an event that might have influenced e' by checking that $C(e_i) \le C(e_{i+1})$ for all i. We emphasize that this does not give us a definite way to tell whether two events are in fact causally related. If it is important to determine conclusively whether events are causally related, one is led to consider the following stronger requirement from a system of logical timestamps.

Definition 3.7. An event-timestamping mechanism satisfies the *strong* clock condition if the following property holds.

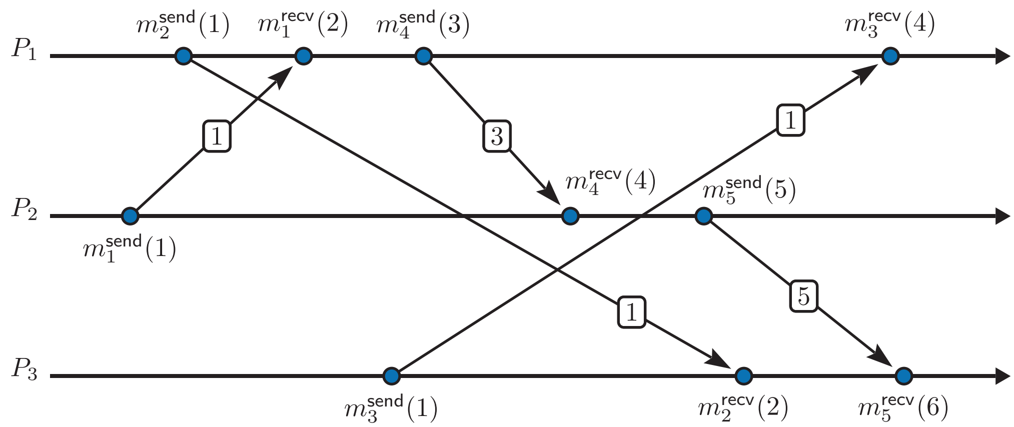
$$\text{For all events } e \text{ and } e', \; e \to e' \iff C(e) < C(e') \qquad \text{(SC)}$$

Note that $\iff$ is notation for "if and only if,"or logical equivalence.

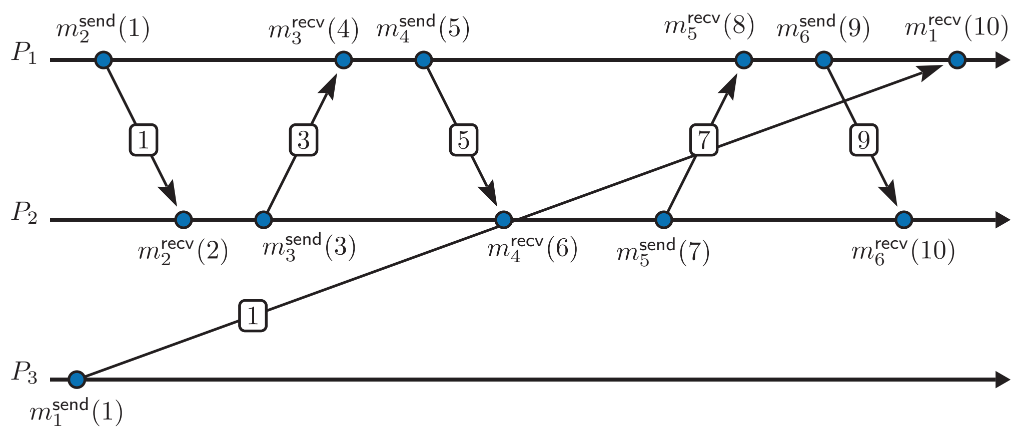
Scalar clocks, below, satisfy (CC), while vector and matrix clocks satisfy (SC).

3.3.1 Scalar clocks

Lamport's scalar clocks [22] require each process P to maintain a single non-negative scalar value C, initialized to 0. The clock follows two simple update rules:



(a) Figure 4a re-depicted with scalar clocks



(b) Figure 4b re-depicted with scalar clocks

Figure 6: Scalar clock examples

R1: Before a message is sent or an internal event occurs, P increments its clock:

$$C := C + 1.$$

The new value serves as the event's timestamp and, for messages, is "piggy-backed" as part of its metadata.

R2: When P receives a message with timestamp C', it updates C as such:

$$C := \max(C, C') + 1.$$

The value is the receive event's timestamp.

Example 3.8. Figure 6 depicts the same events in Figure 4 with scalar timestamps (shown in parentheses) assigned to each event. Piggybacked timestamps are shown as labels on the message arrows.

Scalar clocks satisfy the clock condition (CC). This can be observed by tracing the path of causality between related events and seeing that the clock is incremented at each step. However, they do not satisfy (SC). While e having a lesser timestamp than e' rules out $e' \to e$, it does not imply $e \to e'$. For instance, in Figure 6b, m_1^{send} has a globally minimal timestamp value of 1, but it does not causally precede all events with timestamps greater than 1, or indeed any event except m_1^{recv}.

3.3.2 Vector clocks

The strong clock condition (SC) cannot hold either using scalar clocks or even synchronized physical clocks because they both assign timestamps whose values form a total order, meaning any non-equal timestamps C_1, C_2 satisfy either $C_1 < C_2$ or $C_2 < C_1$. This leaves no way to assign timestamps to synchronous events that satisfy neither $e \to e'$ nor $e' \to e$, except to make their timestamps are equal. However, assigning equal timestamps to logically synchronous events is contradictory, since an event e can be synchronous with multiple events $e', e'' \ldots$ that are not synchronous with each other (recall Example 3.5). The solution is to let timestamps from a partial order, allowing clock values that are not directly comparable.

Vector clocks store one scalar value for each process in the system, which forms a partial order when vectors are compared component-wise. P maintains a vector vc of N non-negative integers, one for each process in $\mathcal{P}$, with all values initialized to 0. To disambiguate P's vector clock from Q's, we sometimes add superscripts, e.g. vc^P versus vc^Q.

The P^{th} component of P's vector clock, denoted $\mathsf{vc}[P]$, is called P's *local time*. For all other processes Q, $\mathsf{vc}[Q]$ represents a lower bound of Q's local time. Vector clocks are updated according to two rules:

R1: Before an internal event occurs or a new message is sent, P increments its local time according to the rule:

$$\mathsf{vc}[P] := \mathsf{vc}[P] + 1.$$

The (entire) updated vc is the event's timestamp and is piggybacked with outgoing messages.

R2: When P receives a message from Q with timestamp vc^Q, vc is updated according to

$$\mathsf{vc}[X] := \max(\mathsf{vc}[X], \mathsf{vc}^Q[X]) \quad \text{for all } X \text{ in } \mathcal{P}.$$

That is, vc is set to the pointwise maximum of the two vectors. After this, P increments its local time:

$$\mathsf{vc}[P] := \mathsf{vc}[P] + 1.$$

The final vector is the timestamp of the receive event.

These rules might be more intuitively understood by demonstration, so we provide figures to demonstrate the vector clock mechanism.

Example 3.9. Figure 7 depicts the same events as Figure 4 with vector timestamps.

For all other Q in $\mathcal{P}$, $\mathsf{vc}^P[Q]$ represents P's conservative estimate of Q's local time, or $\mathsf{vc}^Q[Q]$. This estimate is always a lower bound, since Q's local time may advance without P's knowledge, but P never updates $\mathsf{vc}^P[Q]$ ahead of Q's actual local time. P learns about updates to Q's local time through piggybacked timestamp vectors. This allows P to learn about Q's time without necessarily communicating directly with Q.

Vector timestamps are compared component-wise. This forms a partial order because one vector may be greater than another in some components but less than it in others.

Definition 3.10 (Vector comparison)**.** Let v, w be two vectors. We define the following relations:

$$\begin{aligned} v = w &\iff \forall i, v[i] = w[i] \\ v \preccurlyeq w &\iff \forall i, v[i] \leq w[i] \\ v \prec w &\iff v \preccurlyeq w \text{ and } \exists i, v[i] < w[i] \\ v \# w &\iff \text{ neither } v \prec w \text{ nor } v \succ w \end{aligned}$$

That is, $v \prec w$ if all of w's components are at least as great as v's, and at least one of its components is strictly greater. When two non-equal vectors are compared, and neither is greater than the other, we write $v\#w$ and say the vectors are *incomparable*.

Lemma 3.11. *Vector clocks satisfy the strong clock consistency condition. That is, where $C(e)$ is the vector timestamp of an event, then*

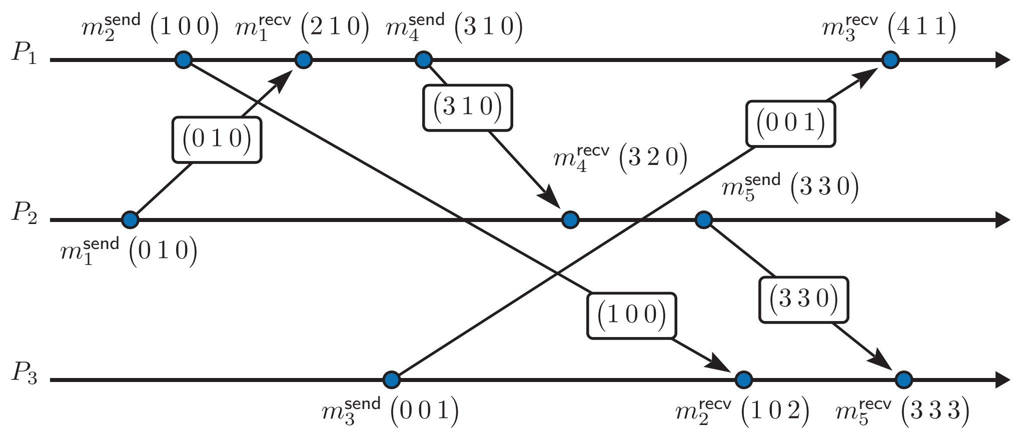
$$e \to e' \iff C(e) \prec C(e').$$

From this it follows that for non-equal events e and e' we have

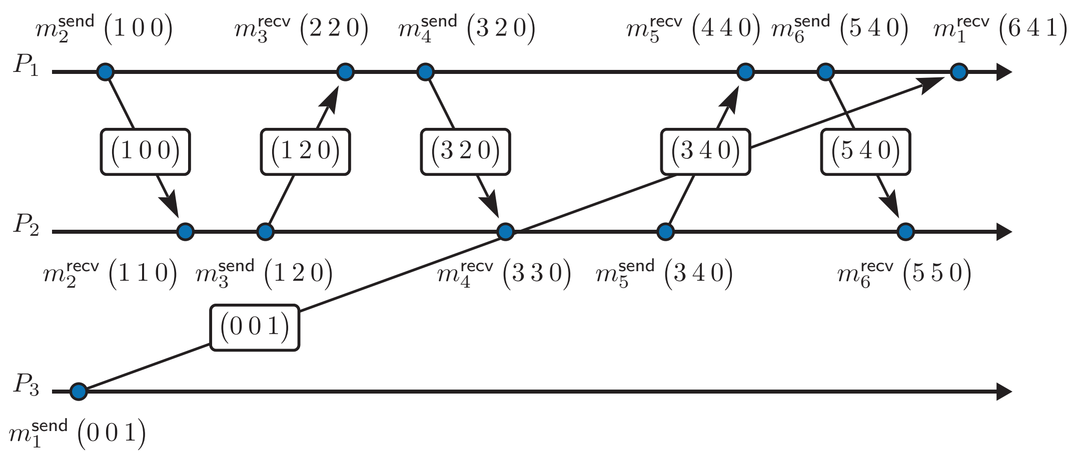
$$e || e' \iff C(e) \# C(e').$$

For reasons of space we omit a proof of the preceding lemma, though the reader may find it enlightening to formalize the details.

Note that vector clocks (and matrix clocks, defined below) both require the processes to agree on the set of members in the group. If groups can change, with members leaving or being added, the timekeeping data structures would similarly have to be updated. We ignore issues of dynamic group membership in this document.



(a) Figure 4a re-depicted with vector clocks



(b) Figure 4b re-depicted with vector clocks

Figure 7: Vector clock examples

3.3.3 Matrix clocks

If a vector clock stores both a local time and a lower bound estimate of every other process's local time, then a *matrix clock* stores a local vector clock and a lower bound estimate of every other process's vector clock. In Section 2, we mentioned the epistemic nature of reasoning about distributed systems: a process can only make decisions based on what it *knows*, which is usually a strict subset of all (system-wide) truths. In many cases, it is important to take into account a kind of second-order knowledge: what we does a process know about what other processes know? In particular, it is often of interest to compute which facts are known to *all* other processes. Vector clocks can be used for this purpose. In some contexts, we must take the idea one step further: what does a process know about what other processes know *about other processes*? This is the purpose of matrix clocks. Sections 4 and 5 feature running examples of mechanisms similar to vector and matrix clocks, called version vectors and matrices. These are used in the context of database replication, where they drive decision-making about which updates need to be propagated and which updates have already been applied everywhere.

In a matrix clock, each process P stores an $N \times N$ matrix mc, initialized to all zeros, with the following interpretation. The row corresponding to P, $\mathsf{mc}[P]$, stores P's vector time. For all other Q, $\mathsf{mc}[Q]$ store P's estimate of Q's vector time. Matrix timestamps are piggybacked with messages, and the receiver uses the sender's vector clock to update their own vector clock as usual, and takes the pointwise maximum of all other rows.

R1: Before a new message is sent, $\mathsf{mc}^P[P][P]$ is updated according to the rule

$$\mathsf{mc}^P[P][P] := \mathsf{mc}^P[P][P] + 1.$$

The entire matrix mc is piggybacked with the message.

R2: When a message is received from Q with a piggybacked matrix mc^Q, mc^P is updated according to two cases

(a) Update the row $\mathsf{mc}^P[P]$ according to

$$\mathsf{mc}^P[P][X] := \max(\mathsf{mc}^P[P][X], \mathsf{mc}^Q[Q][X]) \quad \text{for all } X \text{ in } \mathcal{P}$$

(b) Update all rows for each $R \neq P$ according to

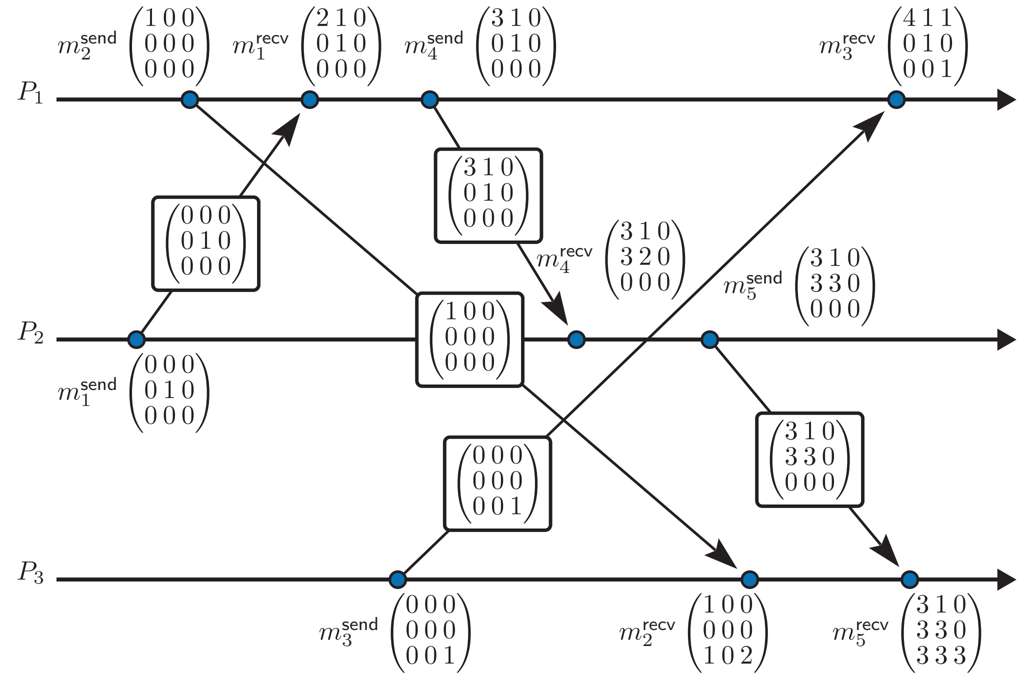
$$\mathsf{mc}^P[R][X] := \max(\mathsf{mc}^P[R][X], \mathsf{mc}^Q[R][X])) \quad \text{for all } X \text{ in } \mathcal{P}$$

After this, P advances its own local time according to the rule

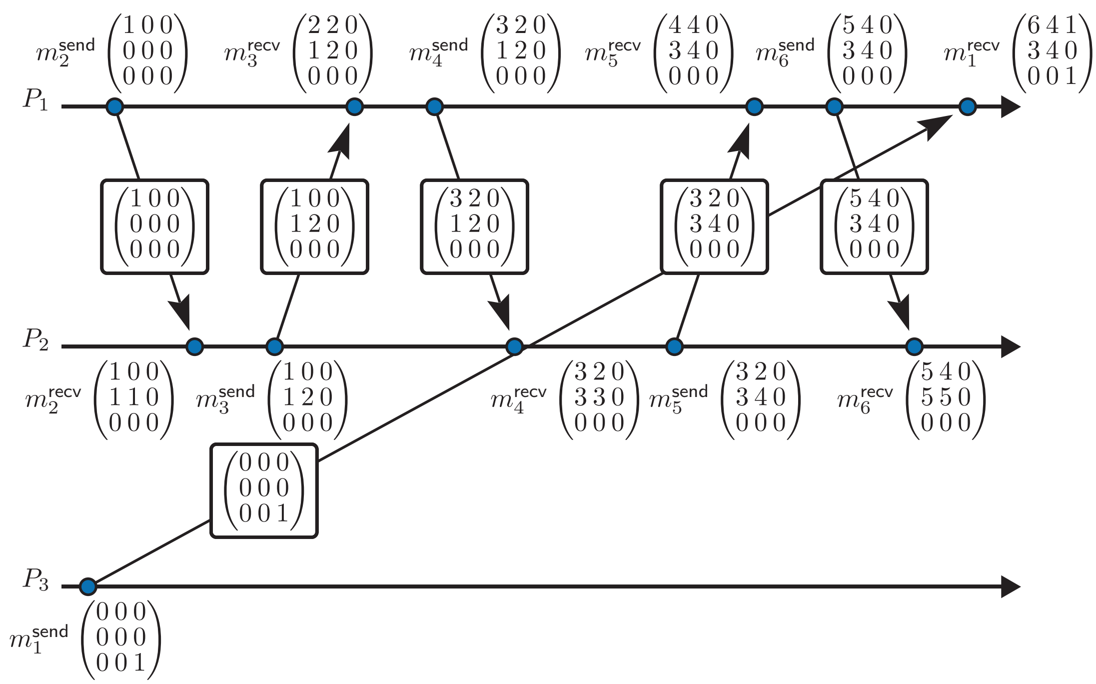
$$\mathsf{mc}^P[P][P] := \mathsf{mc}^P[P][P] + 1.$$

This new matrix is the timestamp attached to the receive event.

Example 3.12. Figure 8 depicts the same events as Figures 4, 6 and 7 using matrix timestamps. By comparison to Figure 7, observe that for each process X, rows of the form $\mathsf{mc}^X[X]$—for instance, the top row of matrices in P_1—act like ordinary vector clocks.



(a) Matrix clock timestamps for the events in Figure 4a



(b) Matrix clock timestamps for the events in Figure 4b

Figure 8: Figure 4 depicted with matrix clocks

3.4 Message Ordering

Coulouris et al. [9] summarized why it is a problem for unpredictable network latencies to cause messages to arrive in a different order than they were sent in.

> "This lack of an ordering guarantee is not satisfactory for many applications. For example, in a nuclear power plant it may be important that events signifying threats to safety conditions and events signifying actions by control units are observed in the same order by all processes in the system."

In this section, we explore different paradigms for message ordering in distributed systems. As with clocks and timestamps, the choice of which ordering guarantee to use depends on the needs of the application. We later generalize the discussion by admitting messages sent to multiple recipients at once, such as in a group chat application, where ensuring predictable message ordering is critical.

When ordering is important, applications do not show messages to the user immediately when they come in from the network—the network can deliver messages in unexpected and undesirable orders, after all. The *arrival* time of a message is when it is received from the network, but instead of acting on it right away, an application may buffer an arrived message while waiting for other messages (such as ones with an earlier causal precedence) to "catch up." When a message is ready to be presented to the user, it is *delivered*. By waiting to deliver some messages, we can ensure the stream of messages in order of their delivery satisfies particular guarantees.

3.4.1 FIFO ordering

A modest requirement is the *first-in, first-out* (FIFO) condition, which stipulates that on any logical communication link between two processes in the system, messages arrive in the order they were sent. The restrictive phrase here is "any logical communication link"—by definition there is one link for any *pair* of processes. Hence, FIFO does not impose any conditions on messages unless they are from the same sender and to the same recipient.

Definition 3.13 (FIFO delivery)**.** The FIFO ordering guarantee is defined by the following condition. Let P and Q be any two processes and m_1 and m_2 be two messages sent from P to P. Then $m_1^{\mathsf{send}} \to m_2^{\mathsf{send}} \implies m_1^{\mathsf{recv}} \to m_2^{\mathsf{recv}}$.

The Internet Protocol (IPv4 or IPv6) by itself does not provide FIFO semantics. In the OSI model, FIFO ordering with reliable delivery is typically provided at the transport layer by the transmission control protocol (TCP).[4] Applications built on top of TCP or a similar transport can therefore take FIFO for granted. Providing FIFO can be as simple as marking messages sent from P to Q with consecutive integers. If message one arrives and then message three arrives, Q infers that message

[4] The other classic internet transport, user datagram protocol (UDP), does not provide any ordering or reliability guarantees.

two is lagging behind, delivering message one to the user, but withholding delivery of message three until after message two is received and delivered.

The guarantees provided by FIFO are minimal because they only apply on a per-link basis: every link requires its own numbering scheme, so message numbers cannot be meaningfully compared across different links. To compare messages globally requires something like causal order, below.

3.4.2 Causal ordering

Causal order is an order guarantee consistent with causal precedence of events. A network provides causally ordered (CO) delivery if it satisfies the following property.

Definition 3.14 (CO delivery)**.** Let P_{dest} be any process and consider all messages m and n sent to P_{dest} (possibly by different senders). CO is satisfied if $m^{\mathsf{send}} \to n^{\text{send}}$ implies $m^{\mathsf{recv}} \to n^{\text{recv}}$. That is, each destination receives messages in an order consistent with causality between their send events.

In mathematical terms, for each process P_{dest}, the function mapping send events to corresponding receive events at P_{dest} must be monotonic with respect to causal precedence. Unlike FIFO, the CO condition enforces a partial order among messages with (in general) different senders.

Example 3.15. Figure 9 demonstrates different message ordering conditions. We make a few observations for emphasis.

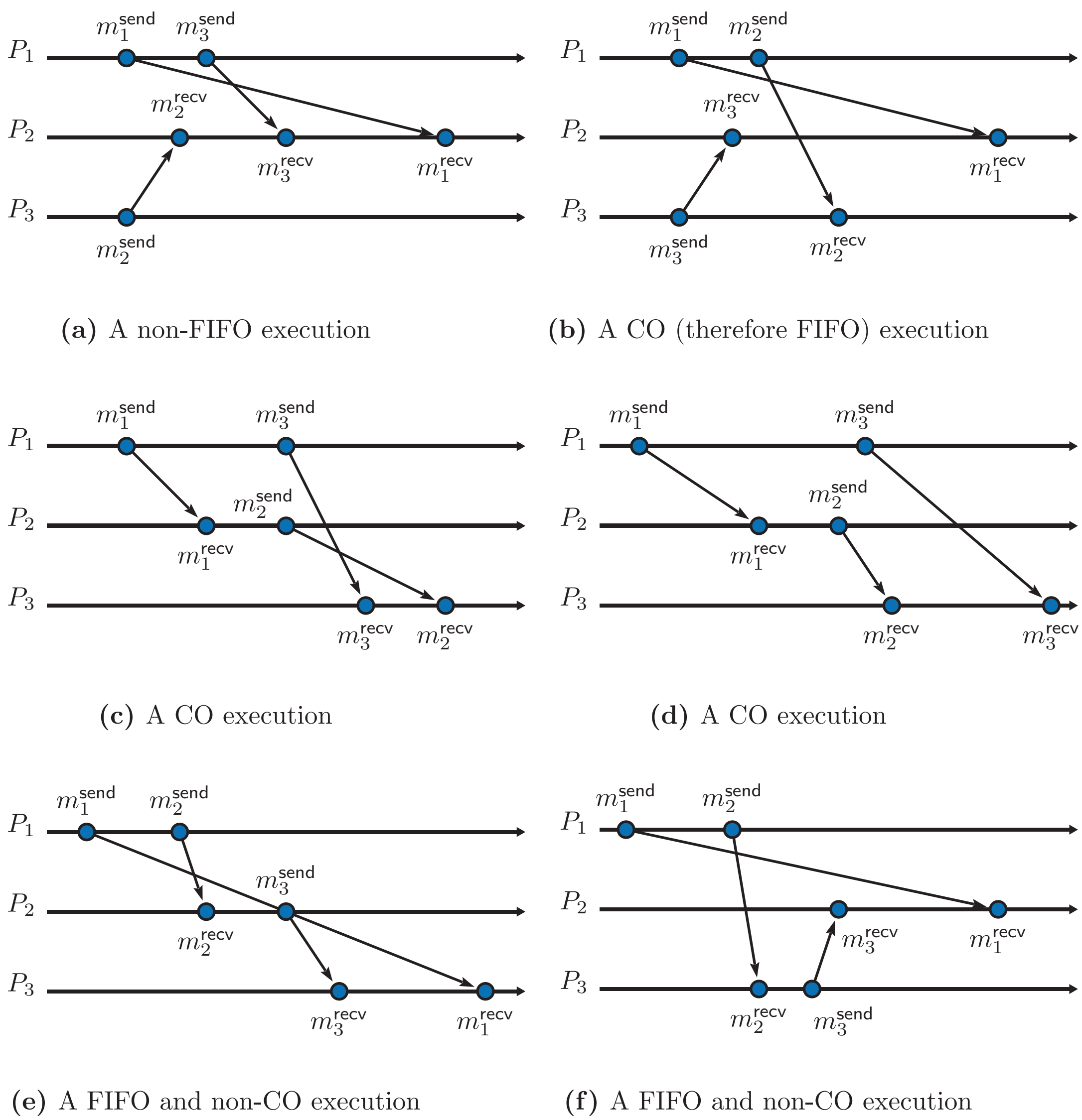
- Figure 9a violates FIFO because messages m_1 and m_2 are both sent from P_1 to P_2, but then arrive in the wrong order.
- Figure 9b satisfies CO and therefore FIFO. Messages m_1 and m_2 arrive in the opposite order but they are sent to different destinations.
- Figure 9e violates CO because the send event of m_1 happens before the send of m_3 via the chain $m_1^{\mathsf{send}} \to m_2^{\mathsf{send}} \to m_2^{\mathsf{recv}} \to m_3^{\mathsf{send}}$, but $m_3^{\mathsf{recv}} \to m_1^{\mathsf{recv}}$.
- Figure 9f violates CO because it is equivalent to 9e with the roles of P_2 and P_3 swapped.

3.4.3 Multicasting and Broadcasting

We now extend the above definitions to the group communication setting by allowing messages to have multiple recipients. For simplicity, we suppose messages are broadcast to all other recipients, though the definitions can easily generalize to "multicast" scenarios where messages are sent to a subset of recipients.

One way to implement broadcasting is to send distinct network messages which, for present purposes, we would treat as a single unit. Alternatively, we can lean on the network itself for assistance, sending a single message specially marked as a broadcast, relying on lower-level protocols in the network to distribute a copy to each recipient. Regardless of implementation, the challenge and importance of ensuring consistent message ordering across an entire group is a paramount concern.

The FIFO and CO broadcast conditions are adapted from Definitions 3.13 and 3.14. Additionally, we present the notion of total ordering (TO).



(a) A non-FIFO execution

(b) A CO (therefore FIFO) execution

(c) A CO execution

(d) A CO execution

(e) A FIFO and non-CO execution

(f) A FIFO and non-CO execution

Figure 9: Message ordering examples

Definition 3.16 (FIFO broadcast)**.** A broadcast primitive satisfies the FIFO semantics if it satisfies the following condition. For any process P, if P broadcasts m_1 before m_2, then all recipients receive m_1 before m_2.

Definition 3.17 (CO broadcast)**.** A broadcast primitive satisfies CO semantics if for any broadcasts m and n, if $m^{\mathsf{send}} \to n^{\mathrm{send}}$, then all destinations deliver m^{recv} before n^{recv}.

In the above definition, the happens before relation is defined just as in the unicast (non-broadcast) setting by following a path of causality along worldlines and message arrows.

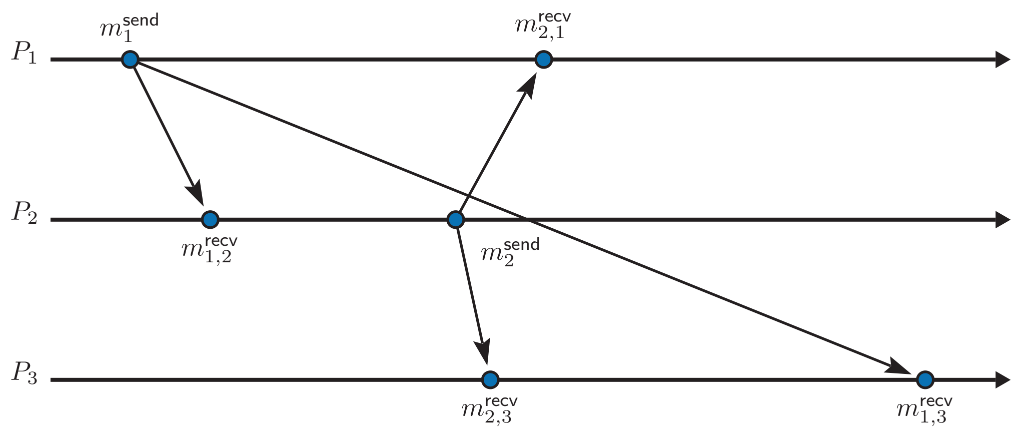


Figure 10: Broadcast example that satisfies FIFO but violates CO

Example 3.18. In Figure 10, causal order is violated. Imagine the following conversation:

P_1 : "I need an ambulance at location A."
P_2 : "Understood, the last ambulance has been dispatched."

However, P_3 receives P_2's response before P_1's request, resulting in this conflicting view:

P_2 : "Understood, the last ambulance has been dispatched."
P_1 : "I need an ambulance at location A."

From P_3's perspective, P_1 appears to be requesting resources that are no longer available. The sort of conflict can lead to confusion, with requests being duplicated or going unanswered. Tracking causal order is crucial to avoid such resource misallocation.

A total order broadcast ensures that all recipients receive the messages in the same order. This order is not required to satisfy any particular constraints except that all recipients agree on it. Such a model is appropriate when it is more important that everyone agrees on a common order of events but the order itself is not necessarily critical.

Definition 3.19 (TO broadcast)**.** For any processes P and Q and messages m and m' that arrive at both destinations, m arrives before m' at both processes or m' arrives before m at both processes.

Total order is independent of causal order, as causality is not total and a total order does not generally respect causality. Thus it sensible to consider also a hybrid notion of *total-causal* order in which all messages are received in a total order respecting causality.

Example 3.20. Figure 11 depicts examples of broadcast message orders.

- Figure 11a trivially satisfies FIFO because no process sends more than one broadcast. CO is violated because $m_1^{\mathsf{send}} \to m_{1,4}^{\mathsf{recv}} \to m_2^{\mathsf{send}}$, while P_2 receives m_2 before m_1. TO is violated because P_2 and P_3 receive the messages in opposite orders.
- Figure 11b violates TO for the same reason above, but satisfies causality because the two send events have no causal relation.
- Figure 11c satisfies TO, but m_2 arrives at both processes before m_1 so FIFO is violated.

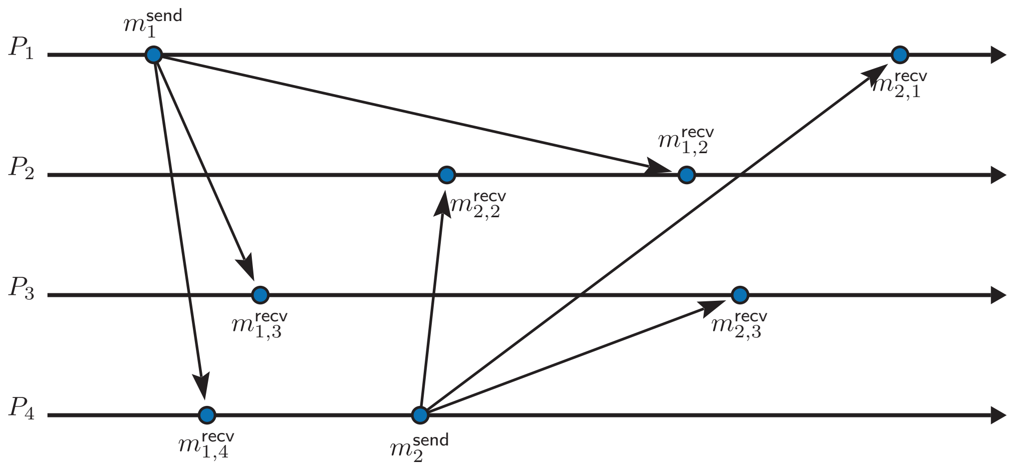
Section 4.5 describes a conventional mechanism used to implement total order broadcast.

Self-delivered messages In some contexts one considers broadcast primitives that include the original sender among the recipients of a message. For simplicity, the examples in this section have not shown this sort of self-delivery, but it is useful in many cases. Self-delivery is useful when combined with ordering guarantees. A typical use case is that participants are using a total order broadcast to maintain local replicas of a state machine that can be advanced by any participating process by announcing updates. This usage is presented in Sections 4 and 5, with the goal of replicating shared state over a disruption-heavy network.

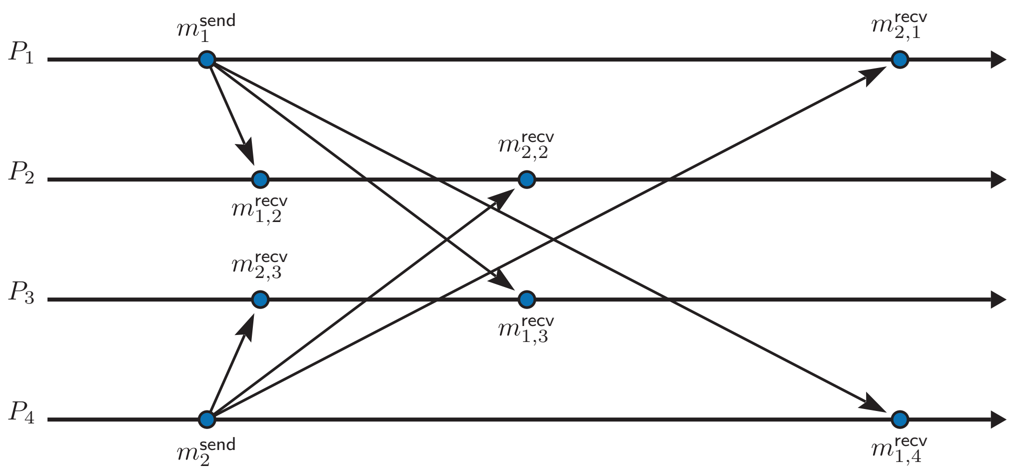
3.5 Shared Memory

Designing a distributed application using direct message passing can be challenging due to the complexity of managing the low-level details surrounding message ordering and reliability. A more abstract approach, the *distributed shared memory* (DSM) framework, simplifies this task by allowing programmers to think in terms of reading and writing to memory locations instead of sending messages over a network.

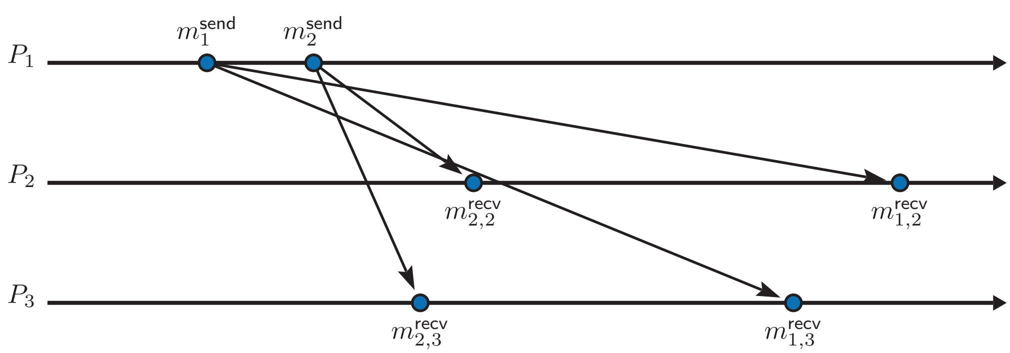
The defining feature of DSM is that it allows all processes to interact as if they had access to a single, unified proof of shared memory—just like processes running on a single computer—despite being spread across different, physically separated computers. This seamless (subject to caveats explained below) experience is facilitated by a middleware layer inside the process called the *memory manager*, which handles all read and write requests submitted by an application. In the background, not directly visible to the application, the memory manager coordinates with other instances over the network to maintain the illusion of a shared state, or what first responders would call a "common operating picture."



(a) Broadcast example that satisfies FIFO but violates CO and TO



(b) Broadcast example that satisfies CO but violates TO



(c) Broadcast example that satisfies TO but violates FIFO

Figure 11: Multicast ordering examples

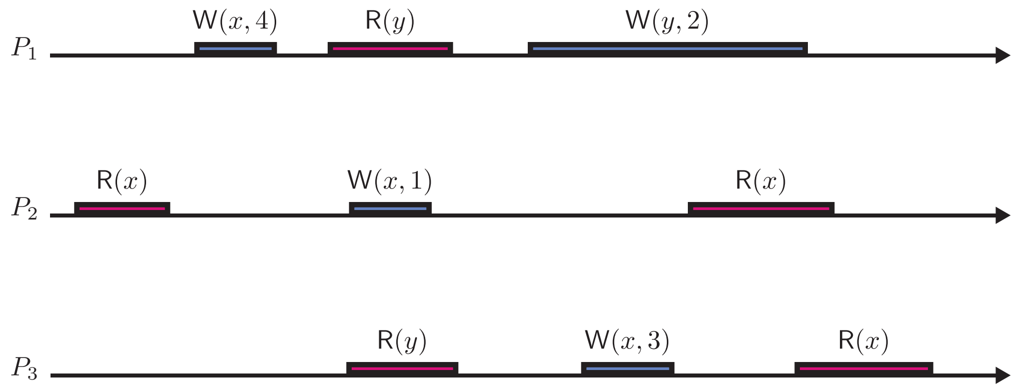


Figure 12: Time diagram for memory operations

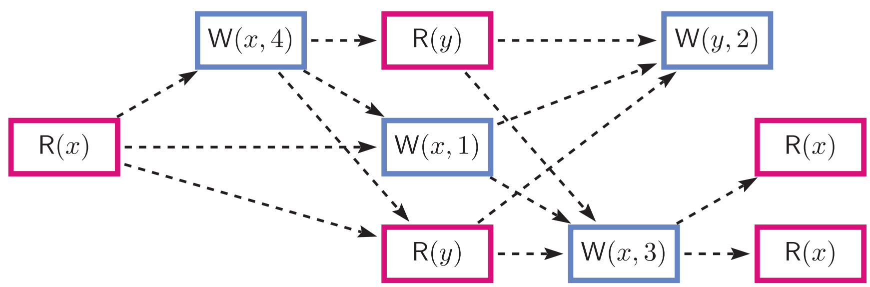


Figure 13: External order relation among operations in Figure 12 (with edges implied by transitivity not shown)

Since there is no free lunch, the seamlessness of the DSM model is subject to caveats, especially this one: usually, a read request handled by the memory manager does not return the most up-to-date value of the memory location it reads. Indeed, in the distributed setting it is not clear a priori what it means for a returned value to be "up-to-date" in the first place, in light of the fact that two different processes can write conflicting values to the same memory location at the same time. For developers, understanding the semantics of the virtual memory layer is a crucial part of building applications that function correctly while providing reliable performance. This requires knowing what guarantees the virtual memory layer makes concerning what values can be returned at which times. Because of the sorts of tensions discussed in Section 2, the design space here is generally marked by a tradeoff between stronger consistency guarantees and faster performance.

Figure 12 depicts an exemplary time diagram for the shared memory abstraction, similar to those for message passing. Two kinds of operations are shown: *reads* and *writes*. A read operation, $\mathsf{R}(x)$, retrieves the value stored at (virtual) location x, which returns some value v. When we want to indicate the value returned by the read, we write $\mathsf{R}(x, v)$. A write operation, $\mathsf{W}(x, v)$, indicates writing v to memory location x, which in code might be written as something like $x := v$.

An operation does not happen instantly, but has a *duration*. An arbitrary read or write operation, Op, spans from the moment of time the operation is invoked by the application ($\mathsf{start}\,(\mathrm{Op})$), to when it finished ($\mathsf{stop}\,(\mathrm{Op})$), returning either the read value or an acknowledgment of the write request. During this span, the memory management layer is usually coordinating in the background with other processes over the network, say by looking up the current value of a memory location, but this is not shown in the diagrams. The entire sequence of requests across all processes forms what we call a *history*. If H is a history and P is a process, we write $H|_P$ to mean just the sequence of operations that happen on P, the so-called *local history* of P.

Because they have a duration, memory operations on different processes, including ones that access the same virtual memory locations, can occur simultaneously. A fundamental relation among operations is their *external order*, the partial order that relates non-overlapping events to their physical times, but does not assign an ordering to events whose executions overlap in physical time.

Definition 3.21 (External order)**.** Let H be a history. An operation Op^1 *externally precedes* operation Op^2 if $\mathsf{stop}\left(\mathrm{Op}^1\right) < \mathsf{start}\left(\mathrm{Op}^2\right)$. This induces an irreflexive partial order on H called *external order*.

The definition states that one operation externally precedes another if it stops before the other is invoked. Note that we are comparing events in terms of real, physical time: external order is the partial order of events witnessed by an outside observer who can watch operations executing globally in real time. Figure 13 depicts the external order among the operations in Figure 12, forming a directed acyclic graph (DAG).

Two events not related by external order are said to be physically concurrent.

Definition 3.22 (Physical concurrency)**.** Consider two operations Op^1 and Op^2. If neither externally precedes the other, in other words if there is some moment in time during which both operations are executing, the operations are said to be *physically concurrent*, written $\mathrm{Op}^1||\mathrm{Op}^2$.

Note that we have reused notation between Definitions 3.4 and 3.22. Though they are similar concepts (both are reflexive and symmetric but generally non-transitive binary relations), physically concurrent memory operations in the DSM model should not be confused with logically synchronous events in the message-passing model.

3.6 Semantics and Consistency

In a sequential application running on a single computer, it is clear how read and write requests should be interpreted. A read request $\mathsf{R}(x)$ should return the most recent value that was written to x by a write operation $\mathsf{W}(x, v)$ (or return a default value if no such write exists, but we will not consider such examples). This is unambiguous because we assume that in a single process, memory operations do not overlap in time, so there is always a sense of which one happened first.

Example 3.23. Consider the following history of operations running inside a single process.

This diagram does not indicate what values are returned by the read operations, but since there is no ambiguity in the order of events, it is clear what these values *should* be. Each read request should return the value (shown in bold below) set by the most recent write to that location.

$$\mathsf{W}(x,0) \to \mathsf{W}(y,5) \to \mathsf{R}(x,\mathbf{0}) \to \mathsf{W}(x,3) \to \mathsf{R}(y,\mathbf{5}) \to \mathsf{R}(x,\mathbf{3})\,.$$

In a distributed system, operations on different processes can run concurrently, so there is no obvious way to arrange events into a total order that all processes can agree on. Consequently, the notion of "most recent" operation is ambiguous, so it does not even make sense to say that read requests always return the most recent written value.

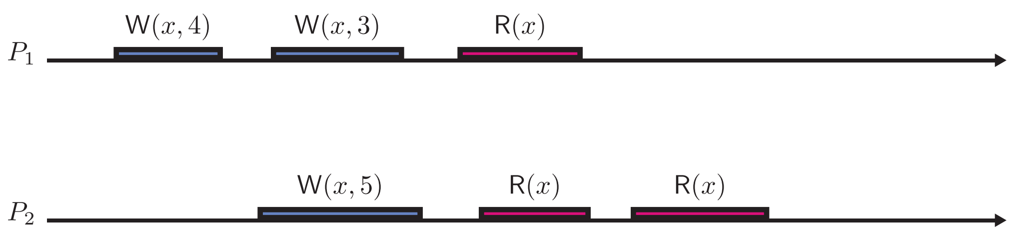


Figure 14: A history with read return values left unspecified, featuring concurrent operations writing to and reading from the same location

Consider the history shown in Figure 14, which contains three operation that write to location x. Two of these operations, $\mathsf{W}(x,3)$ $\mathsf{W}(x,5)$, are executed at overlapping moments in time, making it unclear which should be considered "first." The ambiguity is made concrete by considering the subsequent read operations—which values should they return? Or rather, we should ask which values are they *allowed* to return, since the possibilities are not usually deterministic.

A *memory model* exists precisely to answers questions of the form, "Which values might be returned by read requests in which scenarios?" In the example above, a model would have to answer several questions like the following ones:

1. Do the read operations on P_1 and P_2 have to return the same value?

2. Can the second read operation at P_2 return a different value than the first one?

3. Is it ever possible for any of the $\mathsf{R}(x)$ operations to return the value 4?

One can consider many different ways of answering these questions, depending on the consistency requirements of the high-level application. The strictest memory model is called *linearizability*. If the distributed system whose history is shown in

Figure 14 is linearizable, the all three read operations must return the same value, and this must be either 3 or 5. This model is intuitive but too strict for our use case, so programmers must be prepared for less rigidly prescribed behavior from the memory manager.

Choosing to implement a particular memory model requires balancing the needs and expectations of the application against its performance characteristics, including its usage patterns and networking environment. An application designed for one memory model would generally misbehave, often in a way that is difficult to diagnose, if executed in an environment that implements a different model. However, on the other hand, implementing a stricter memory model may impose a prohibitive overhead on application performance.

To resolve the ambiguity caused by overlapping memory operations, one might attempt to assign physical timestamps to them and use this to define an agreed-upon global total order of operations. However, using these kinds of orders in practice requires sufficiently fine-grained timestamps from synchronized clocks. For our environments this is often infeasible (see Section 3.1).

3.7 Strong Consistency Models

This section considers the two major memory models usually said to provide "strong" consistency: linearizability and sequential consistency. Both models involve the notion of a sequential history, or a set of operations arranged into a particular total order.

Definition 3.24 (A sequential history)**.** A *sequential history* is a set of memory operations in a particular total order. If H is a history, a *serialization* is any total order among the operations in H.

Linearizability and sequential consistency both require that all read operations return values consistent with some serialization of the global history. That is, they stipulate that among all the operations in H, there is some way of ordering them so that all read operations return the values of the most recent write operations. Where the two models differ is in how they constrain which serializations of H are allowable.

3.7.1 Linearizability

Linearizability can be concisely defined as a system that acts like "each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response." [17] The same condition is known (though sometimes with subtle variations in meaning) by names like atomic consistency and external consistency. It means almost the same thing as strict serializability, except the latter terminology is used to discuss transactional databases and implies other database-related guarantees. Formally, a linearizable history is defined by three features.

Definition 3.25 (Linearizable history)**.** Let H be a history of memory operations. H is *linearizable* if it satisfies the following three rules.

1. **Global Agreement on Order**: All processes behave (defined below) as if they are observing the same serialization of H, meaning some particular total order σ.
2. **Correct Responses**: Each read request $\mathsf{R}(x, a)$ returns the value of the most recent write request $\mathsf{W}(x, a)$ as ordered by σ.
3. **Consistent with External Order**: The serialization of H is consistent with external order: if $\mathsf{stop}\left(\mathrm{Op}^1\right) < \mathsf{start}\left(\mathrm{Op}^2\right)$, then σ also orders Op^1 before Op^2.

Definition 3.25 is concerned with an individual history of some distributed application. When the application only ever admits linearizable histories, as permitted by the memory manager, the entire system is said to be linearizable.

Definition 3.26 (Linearizable system)**.** A DSM application is linearizable if all possible histories of the application satisfy Definition 3.25.

Consider the history Figure 14 again. If the application is linearizable, the read responses must agree on some order σ consistent with external order, implying that $\mathsf{W}(x, 4)$ happens before $\mathsf{W}(x, 3)$ and $\mathsf{W}(x, 5)$, but the latter operations can occur in any order. Since P_1 and P_2 have to agree on this order, all three read responses will return the value written by whichever write is ordered last (meaning most recent), which is either 3 or 5. These possibilities are illustrated in Figure 15.

Figure 15 also depicts an equivalent, more visually intuitive way, of approaching defining linearizability. A linearizable history where returned values are consistent with a choice of linearization point for each operation.

Definition 3.27. A *linearization point* t for an operation Op is a time in the range $[\mathsf{start}\,(\mathrm{Op})\,, \mathsf{stop}\,(\mathrm{Op})]$, between the operation's invocation and response. We forbid distinct operations from having an overlapping linearization point.

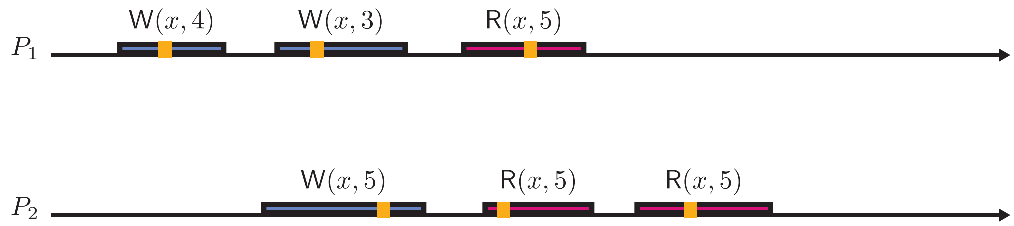
A history is linearizable if there is some choice of linearization point for each operation, and returned values are consistent with operations taking effect in whole and instantaneously at their linearization points. Figure 15 depicts possible choices of linearization points in yellow.

3.7.2 Sequential consistency

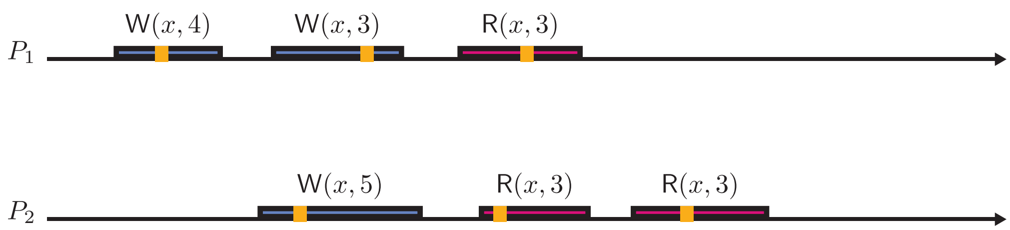
Linearizability offers very strong guarantees related to real-time constraints, but for many applications this requirement is a burden for performance. A more relaxed model, sequential consistency, provides comparably strong guarantees but does not impose the same constraints with respect to external order. Whereas linearizability requires operations to be consistent with a serialization respecting external order, sequential consistency allows any serialization that respects *program order*.

Definition 3.28 (Program order)**.** An operation Op^1 precedes another operation Op^2 in *program order* if the events occur in the same process P and $\mathsf{stop}\left(\mathrm{Op}^1\right) < \mathsf{start}\left(\mathrm{Op}^2\right)$. In this case we write $\mathrm{Op}^1 \underset{P}{\to} \mathrm{Op}^2$.

If two operations occur in different processes, they are not related by program order. That is the distinction between program and external order.



(a) A linearization where the read operations return 3



(b) A linearization where the read operations return 5

Figure 15: Two possible linearizations of Figure 14 with linearization points shown in yellow

The definition of sequential consistency follows the same structure as that for linearizability, with program order in place of external order.

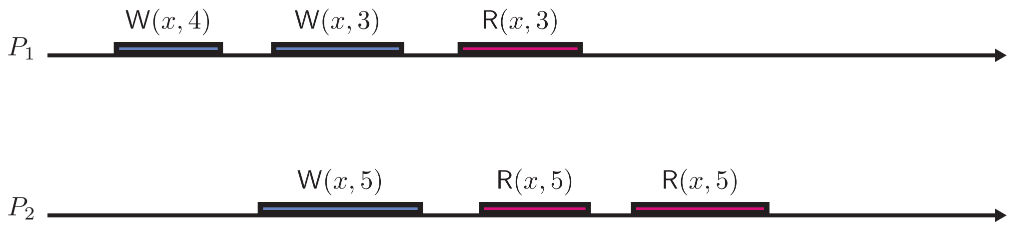
Definition 3.29 (Sequentially consistent history)**.** An history H is *sequentially consistent* if the following three rules are satisfied.

1. **Global Agreement on Order**: All processes behave as if they observe the same serialization σ of H.
2. **Correct Responses**: Read requests return the value of the most recent write request to the same location according to σ.
3. **Consistent with Program Order**: σ is consistent with program order: if $\mathsf{stop}\left(\mathrm{Op}^1\right) \underset{P}{\to} \mathsf{start}\left(\mathrm{Op}^2\right)$ for some P, the serial history must include Op^1 before Op^2.

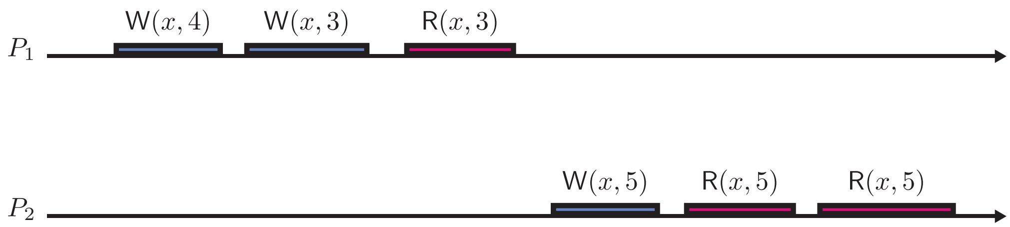
Since external order imposes more constraints than program order, linearizable histories, like those shown in Figure 15, are always sequentially consistent.

Lemma 3.30. *A linearizable execution is sequentially consistent.*

The converse of Lemma 3.30 does not hold, meaning some sequentially consistent executions are not linearizable. As noted earlier, there are only two linearizable histories of Figure 14, and they both require all three read operations to return the same value. Examples 3.31 and 3.32, below, demonstrate sequentially consistent histories where P_1 and P_2 read non-equal values.



(a) Sequentially consistent history where P_1 and P_2 read different values of x



(b) A consistent serialization respecting program order

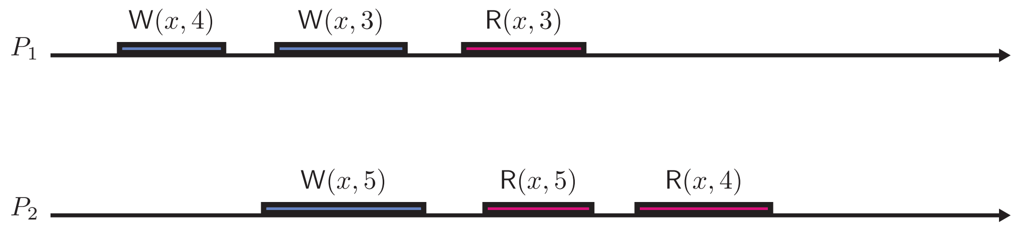
Figure 16: A sequentially consistent history and its consistent serialization

Example 3.31. Figure 16a depicts a sequentially consistent history of the operations depicted in Figure 14. This history is non-linearizable because P_1 and P_2 read different values for x. It is sequentially consistent because it returns values consistent with the alternate history shown in Figure 16b, which is sequential because has no overlapping operations. The latter can be obtained by "sliding" the operations in P_2 along their worldline so they occur after those in P_1.

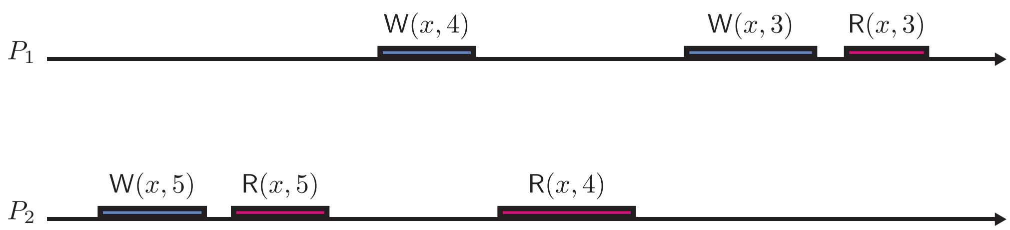
Example 3.32. Figure 17a depicts another sequentially consistent history of the operations in Figure 14. In this example, P_2 appears to travel backwards in time, reading the stale value 4 immediately after reading 5. This history is consistent with the serialization shown in Figure 17b.

The time-traveling nature of Example 3.32 can be explained by remembering that the DSM model is implemented in terms of message-passing. P_2 may read a stale value of 4 because of the time it takes for the memory manager running on P_1 to notify P_2 about the $\mathsf{W}(x,4)$ operation. This notification may not be received until after P_2 performs the $\mathsf{R}(x,5)$ operation.

Sequential consistency is an intuitive property for reasoning about the possible behaviors of distributed programs. Note that each process in the system issues memory operations in a particular order—these can be thought of as individual steps in a program. Before the application is executed on real computers, there is no guarantee about the relative timing of program steps that run on different computers, since different machines may run at different speeds. For instance, before running the program and observing the series of events shown in Figure 17a, we did not necessarily know that the $\mathsf{W}(x,4)$ operation would precede the $\mathsf{W}(x,5)$ operation



(a) Sequentially consistent history where P_2 reads x with value 4



(b) A consistent serialization respecting program order

Figure 17: A sequentially consistent history and its consistent serialization

in real-time—they are both the first steps of their respective programs, with no relation to each other. The alternate order of events shown in 17b, where W$(x,5)$ precedes W$(x,4)$, is just as likely a priori as the one that was actually observed. Sequential consistency guarantees each program is always consistent with one of the serializations that can be expected a priori, before the program is executed and a real-time external order of events is fixed.

3.8 The CAP Theorem

Real-world systems rarely function as a perfectly coherent, unified system. One fundamental gap between idealized and real-world behavior stems from a well-understood and fundamental tradeoff between coherence and performance. The more "coherence" we demand from the system, the more processes have to communicate over the network, whose unpredictable delays impose overheads that degrade performance. Conversely, the more we demand immediate answers from our system, the less time a process has to communicate with other processes, so the system as a whole does not seem as coherent and unified to end users.

This tradeoff is made fully stark by considering the possibility that the network suffers from a partition, which prevents some processes from communicating with others.

Definition 3.33 (Network partition)**.** A *network partition* is a span of time where some nodes are unable to communicate with another set of nodes on the network.

In 1999, Fox and Brewer [13] articulated a formal tradeoff between three desirable properties of distributed systems: consistency, availability, and an ability to function

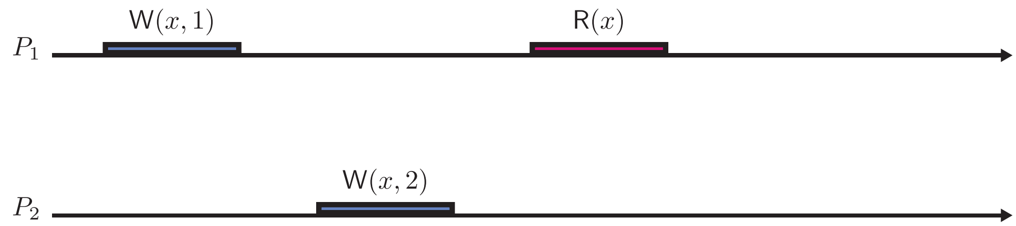


Figure 18: A history where linearizability cannot maintained during a network partition

during network partitions. This observation was formalized and rigorously proven by Gilbert and Lynch in 2002 [14]. Despite its prominence at the heart of distributed systems, and the fact that its proof is fairly straightforward, the CAP theorem is sometimes misunderstood, so it is worth clarifying its key terms.

Consistency: Gilbert and Lynch define consistency as linearizability.

Availability: A CAP-available system eventually responds to every client request (a read or write operation) after a finite time.

Partition tolerance: A partition-tolerant system continues to function in the face of arbitrary partitions in the network.

In the last case, the possibility is allowed that a partition never recovers. This could happen if a critical communications cable is permanently severed, for instance.

The CAP theorem is the simple observation that a distributed system cannot guarantee all three properties simultaneously: a system that operates during network partitions cannot ensure both linearizability and availability. We give only the informal sketch here, leaving the interested reader to consult the more formal analysis by Gilbert and Lynch. The key assumption in the proof is that a process's behavior is only affected by the messages it receives. During a partition, its behavior is the same regardless of what other processes do, since it does not communicate with them and cannot be affected by them. Below, we term this property *behavioral invariance.*

Theorem 3.34 (The CAP Theorem)**.** *If indefinite network partitions are possible, then a distributed system cannot guarantee both linearizability and eventual availability.*

Proof. Consider the history in Figure 18. Clearly there is only possible linearization of this history: $\mathsf{W}(x,1)$ executes, $\mathsf{W}(x,2)$ executes, and $\mathsf{R}(x)$ reads the value 2. Now suppose P_1 and P_2 are separated by a network partition, meaning P_1 does not receive any messages from P_2. This leaves two possibilities for how P_1 might handle the $\mathsf{R}(x)$ operation:

1. P_1 can proceed despite the network partition and return a value. By behavioral invariance, P_1 would have to return $\mathsf{R}(x,1)$. This is value it would read (by

linearizability) if P_2 did not write to x. Behavioral invariance means P_1's reads the same value regardless of what P_2 does (since in either case, it receives no communication from P_2). Returning 1 violates linearizability in the case above where P_2 *does* write to x.

2. P_1 might detect (or assume) the network suffers a partition and refuse to handle the $\mathsf{R}(x)$ until it is able to communicate with P_2 again, whereupon it would learn the correct value of x is 2. However, we do not assume the partition has to recover, in which case P_1 waits forever, which violates availability.

Thus, we cannot have both linearizable consistency and availability. More precisely, to ensure both of these properties, we would have to assume the network never suffers from partitions, but this is unrealistic. As discussed in Section 2.2.1, real-world examples of partitions in wildland firefighting environments are common. □

The proof above raises the question of whether the weaker notion of sequential consistency can be used to avoid being subject to the CAP theorem. The answer is negative: sequential consistency is also CAP-unavailable.

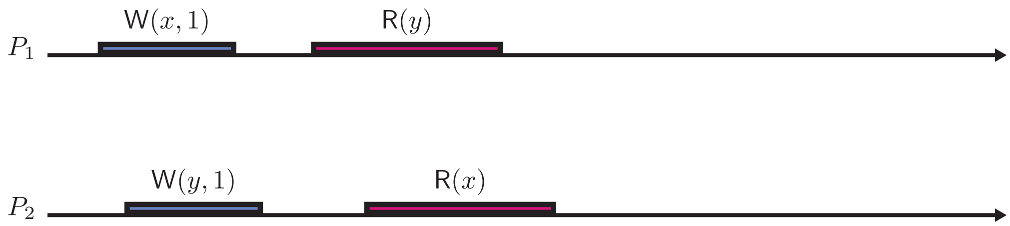
Lemma 3.35 (CAP for sequential consistency)**.** *An eventually-available system cannot provide sequential consistency in the presence of network partitions.*

Proof. Consider the history in Figure 19a and suppose all memory locations are initialized to 0. Following similar reasoning as above, behavioral invariance means that if P_1 and P_2 are separated by a partition and remain available, P_1 has to return 0 to the request $\mathsf{R}(y)$—that is the value it would return assuming P_2 does not write to y. Likewise, P_2 would read $\mathsf{R}(x, 0)$. However, the resulting history, shown in Figure 19b, is not sequentially consistent. For $\mathsf{R}(y, 0)$ to be consistent, the sequential order of operations would have to arrange $\mathsf{R}(y, 0)$ before $\mathsf{W}(y, 1)$—otherwise reading 0 is incorrect. This results in the order shown in Figure 19c, which is incorrect because $\mathsf{R}(x, 0)$ occurs after $\mathsf{W}(x, 1)$. The situation is symmetric: to order $\mathsf{R}(x, 0)$ before $\mathsf{W}(x, 1)$ results in an order where $\mathsf{W}(y, 1)$ precedes $\mathsf{R}(y, 0)$ (Figure 19d). □

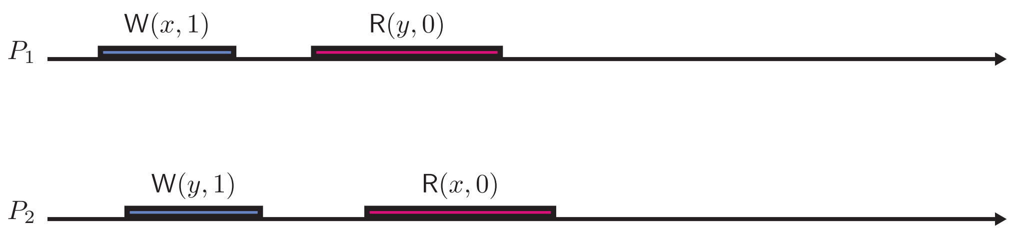
3.8.1 Consequences of CAP

While the CAP theorem is theoretically simple, its implications are more nuanced than they may appear [5]. A common oversimplification is that the CAP theorem represents a "choose 2 of 3" scenario: a system designer can choose at most two of consistency, availability, and partition resilience. In fact, real systems may balance weaker forms of all three properties. The CAP theorem only rules out the combination of all three properties when each of them is defined in an idealized, rigid way.

In practice, applications often settle for weaker levels of consistency than linearizability or sequential consistency. Resilience to network partitions typically requires coping with intermittent, rather than indefinite, communications failures. Finally, availability is best measured in terms of actual response time as experienced by the user, and not the mere assurance that a request will "eventually" be handled. Thus, each of these dimensions is actually quantitative in nature, rather than an all-or-nothing proposition.



(a) An execution that cannot maintain sequential consistency during a network partition



(b) By behavioral invariance, if there is a network partition, the values read for x and y must be their initial values of 0

(c) A serial order where $\mathsf{R}(y, 0)$ precedes $\mathsf{W}(y, 1)$ forces $\mathsf{W}(x, 1)$ to precede $\mathsf{R}(x, 0)$, violating sequential consistency

(d) A serial order where $\mathsf{R}(x, 0)$ precedes $\mathsf{W}(x, 1)$ forces $\mathsf{W}(y, 1)$ to precede $\mathsf{R}(y, 0)$, violating sequential consistency

Figure 19

The locality principle is also highly relevant when considering implications of the CAP theorem for a real system. The closer agents are located, the more reliable their communications will be in general, and the more applications can provide consistency and availability for operations that only require coordinating with nearby agents. At short time scales, operations that only require local coordination are common.

3.9 Causal Consistency

Causal consistency, defined by Ahamad et al. [1], is a weaker memory model than sequential consistency. Whereas sequential consistency requires the system as a whole to behave as if all write operations take place in some total order (which must also respect program order), causal consistency allows different processes to behave as if they witnessed past write operations take effect in different orders. Only write operations related by *causal precedence* are required to take effect in a common order across all processes: "reads respect the order of causally related writes." [1]

We have not defined what causal precedence means for memory operations. The notion is similar in its motivation to causal precedence in message-passing framework (Definition 3.2) and the idea of causal broadcast (Definition 3.17) but the definition of causally related memory operations is not as simple as that of causal precedence among messages. In message passing, a receive event is always associated with a unique send event, but multiple processes can write the same value to the same memory location, and for a later operation that reads this value, it is not clear which write "caused" it. For this purpose we define a *writes-into* order.

Definition 3.36 (Writes-into order)**.** Let H be a history of memory operations.[5] A "writes into" order $\mapsto$ is a binary relation where each read $\mathsf{R}(x, v)$ is paired with the write operation that determined the value that operation read. That is, all related pairs are of the form $\mathsf{W}(x, v) \mapsto \mathsf{R}(x, v)$ and every read is paired with some write.

Ahamad et al. [1] give a slightly more complex definition to allow for operations that read uninitialized memory locations. For simplicity we assume each memory location is written to before it is read.

Definition 3.37 (Causality order on memory operations)**.** For a given writes-into order $\mapsto$ on H, the associated *causality order* $\rightsquigarrow$ is the transitive closure of the union of $\mapsto$ and program order. That is, if $\mathrm{Op} \rightsquigarrow \mathrm{Op}'$, then one of the following holds:

- $\mathrm{Op} \underset{P}{\rightarrow} \mathrm{Op}'$ for P
- $\mathrm{Op} \mapsto \mathrm{Op}'$
- There is some Op'' such that $\mathrm{Op} \rightsquigarrow \mathrm{Op}''$ and $\mathrm{Op}'' \rightsquigarrow \mathrm{Op}'$

By fiat, we also require that $\rightsquigarrow$ must not be cyclic, meaning their are no "causality loops" like $\mathrm{Op} \rightsquigarrow \mathrm{Op}' \rightsquigarrow \mathrm{Op}$. If $\mathrm{Op} \rightsquigarrow \mathrm{Op}'$ in a causality order, we say Op causally precedes Op'.

[5] We are treating H as a *multiset*, meaning for example that separation operations both of the form $\mathsf{W}(x, v)$ are considered distinct.

We can now define causally consistent executions of memory operations. For a history H and each process P, let $A|_P$ be the union of $H|_P$ and the set of all writes in H.

Definition 3.38 (Causal consistency)**.** An execution is causally consistent if each P behaves as if it observes some serialization of $A|_P$ consistent with $\rightsquigarrow$.

Notably, Definition 3.38 does not require that all processes behave as if they are observing the *same* serialization.

Example 3.39. Figure 20a depicts a causally consistent execution of the operations in Figure 14 that is not sequentially consistent. The writes-into order is depicted with dotted edges. The operations $\mathsf{W}(x, 5)$ and $\mathsf{W}(x, 3)$ are not related by $\rightsquigarrow$ and therefore do not have to appear to take effect in the same order at all processes.

Suppose this history were consistent with some serial order. By correctness, $\mathsf{W}(x, 5)$ must be the last write before $\mathsf{R}(x, 5)$. By program order, $\mathsf{W}(x, 5)$ comes before $\mathsf{R}(x, 3)$. By correctness, $\mathsf{W}(x, 3)$ must come before $\mathsf{R}(x, 3)$, but not before $\mathsf{W}(x, 5)$. But then it must be the last write before $\mathsf{R}(x, 5)$, a contradiction.

Causal consistency is not subject to the limits of the CAP theorem.

Lemma 3.40 (Causal consistency is CAP-available)**.** *A system can enforce causal consistency during network partitions.*

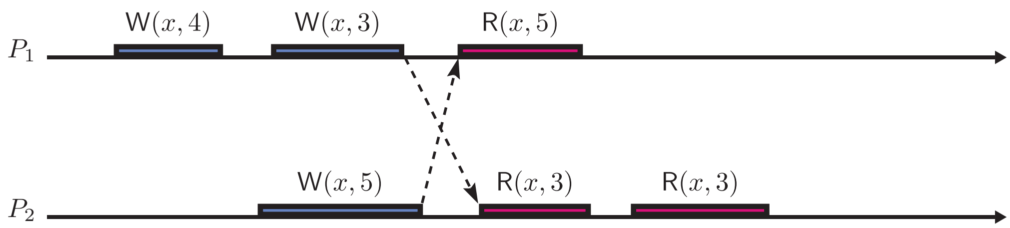
Proof. Consider two processes that execute read and write operations purely locally. That is, they never send messages to other processes, and they always read the most recent value they have written to a location, regardless of what other processes do.

This situation is always causally consistent for the causality relation generated by the empty writes-into order. If the writes-into order is empty, each process only has to be consistent with some serial order of write operations, and processes do not have to agree on this order. In particular, each P is consistent with a serialization where all write operations issued by other processes are executed *after* all of P's own read operations. $\square$

The proof of the Lemma 3.40 speaks to the weakness of causal consistency. In the proof, different processes are allowed to deviate arbitrarily far from consistency with each other. At any moment in time, they may diverge wildly, which violates our goal of maintaining a common operating picture. Causal consistency is too weak to place an upper bound on the divergence observed by users, which suggests it is not the most appropriate model for the kinds of safety-related applications we have in mind. Section 5 will consider how a continuous consistency model can provide these sorts of bounds.

3.10 Section Summary

This discussion has explored the key challenges involved in building distributed systems that connect geographically dispersed components over unpredictable networks. The variability in message delays, particularly in the context of broadcasts sent



(a) A causally consistent history with a writes-into order shown with dotted edges

(b) A serialization consistent with the values read by P_1

(c) A serialization consistent with the values read by P_2

Figure 20: A causally consistent but sequentially inconsistent history

to multiple recipients, can result in messages that arrive in different orders. This situation can lead to chaos if a message-ordering discipline is not imposed.

To mitigate the effects described above, distributed systems must track the causal precedence relation between events. Because physical clocks are not generally reliable enough for this purpose, especially at fine time scales, logical clocks—scalar, vector, and matrix clocks—are typically used, each with a different tradeoff in terms of precision of the information tracked and the administrative and messaging overhead. If groups can change dynamically, as in our use cases, then additional group membership protocols are needed to ensure that all processes know which other processes are participating in the system at any moment.

Programmers may find it easier to frame distributed applications in terms of reading and writing from a shared pool of virtual memory, rather than sending messages, using distributed shared memory framework. The fact that processes can access the same virtual memory locations at the same time makes it challenging to maintain systemwide coherence. Strong consistency models like linearizability and sequential consistency provide the illusion of a single source of truth, but the CAP theorem makes it virtually impossible to guarantee these properties over connection-challenged networks. Weaker models like causal consistency are not subject to the same limitations, but they do not enforce limits bounding how far apart data replicas can diverge, rendering them potentially unsuitable for safety-related applications.

4 Timestamped Anti-Entropy

This section presents Golding's Timestamped Anti-Entropy (TSAE) protocol [15], a *weak consistency group communication* mechanism that provides a form of multicasting to applications. We picture TSAE as a key driver of communication across a distributed system supporting first responders across a region or state during a wide-area event, like the kind discussed in Section 2. Such a system might track firefighters' locations, monitor a wildfire's boundaries, disseminate weather data, and orchestrate resource deployment. In this context, TSAE would have responsibility for ensuring that every update eventually reaches each of its intended recipients.

Golding's thesis [15] presents TSAE as one component of a broader toolkit for developing distributed applications. Before laying out the protocol, it is useful to explain what role it is meant to have in the context of a larger distributed system. We imagine a group of cooperating processes, referred to by Golding as "principals," managing shared state and communicating through message exchange. These messages are an application-specific construct and might contain things like instructions to update a database. These may not map one-to-one with low-level network messages like those discussed in Section 3.2, which we think of as analogous to Internet Protocol (IP) packets. Note that an IP packet might contain multiple application messages, and a large application message may be split up into multiple packets during network transit.

At a lower level, we assume the network supports point-to-point (unicast) communication, but it does not have to guarantee FIFO ordering and may duplicate network messages, though it does not spontaneously create new ones. The network may not be reliable, but we assume that persistent attempts to communicate with a distant node will eventually succeed. The network might support native multicasting of network-level messages, which could be used as part of an implementation of TSAE, but the TSAE layer itself provides a form of application-level multicasting regardless of the network infrastructure.

A typical use case of TSAE is to maintain replicas of a shared database at multiple sites, for which purpose its reliable eventual delivery makes it well-suited as a fault-tolerant messaging component. Here, "database" is used broadly to refer to any data store updated according to some logical model, such as a relational schema, a key-value store, or a document store. Processes update the data by broadcasting update messages using TSAE. Updates are eventually delivered to every process, at which point recipients apply it their own replica. Of course, the semantics of updates are decided by the application. If the underlying data model requires updates to be applied in the same order at all replicas, TSAE can be paired with a totally ordered delivery component; this provides a general mechanism for replicating a state machine. Timestamped anti-entropy with totally ordered delivery is described in Section 4.5 and forms the basis of the distributed memory consistency protocol in Section5.

Replication is an alternative to managing state in a central location, such as a datacenter. This provides resiliency by removing single points of failure. It supports scalability by allowing any process with a replica to deliver services to clients, distributing the load. Finally, replication exploits locality (introduced in Section 2.2) because by servicing user requests at the nearest replica, we can rely on communications links that are generally faster and more reliable.

One highly relevant use case of globally replicating state is to support offline usage of applications. This represents taking locality to its limit. Responders in the field may not be able to connect to a datacenter, but in the meantime they can interact with the application on their device backed by a local replica of the data. This is useful for as long as the agents can be confident that their replica has not diverged too far from the rest of the system. A key point is that this is an application-level matter. How quickly shared state evolves, how to quantify divergence, and how far apart is "too" divergent are determined by user intent, usage patterns, and operational environment, among other things. Because updates are merely delivered *eventually*, TSAE does not bound the divergence of each replica from the "ideal" value of the shared state at any moment. Following Yu and Vahdat [42], we show in Section 5 how an implementation of the conit (consistency unit) model on top of TSAE can be used to enforce, through so-called compulsory anti-entropy sessions, quantitative consistency requirements separately for each replica.

We now present TSAE in terms of its assumptions (4.1), data structures and invariants (4.2–4.4), ordering component (4.5), and storage recycling procedure (4.6). Finally, we discuss how version matrices can be used instead of the loosely synchronized physical clocks assumed below (4.7).

4.1 Assumptions

We assume each process is associated with an identifier. In practice, this might be a Uniform Resource Identifier (URI) [3] following a structured format such as `<agent-id>@<agency>.<state>.<country>`. For instance, the first author's father, Firefighter 116 with the Orange County Fire and Rescue Department in Florida, might be assigned this fictional identifier:

`firefighter116@ocfrd.fl.us`

In the text, we will not make a distinction between a process P and its identifier.

The first assumption we make for the presentation of TSAE is as follows:

> **Static process group**: The set of group members is a fixed set of processes $\mathcal{P} = \{P_1, P_2, \ldots P_n\}$ where the identifier of each process is known to all of them.

Golding's thesis describes how to connect TSAE with a dynamic group membership management component, but we do not describe it here for simplicity.

The second assumption we make is that processes have nearly synchronized clocks. We write $\mathsf{clock}\,(P, t)$ for the clock value at P at real time t.

> **Loose clock synchronization**: Processes' physical clocks are approximately synchronized, meaning the clocks of any two processes differ by at most some small constant δ:
>
> $$\forall t,\, \forall P, Q \in \mathcal{P}, |\mathsf{clock}\,(P, t) - \mathsf{clock}\,(Q, t)\,| < \delta$$
>
> The clock resolution must be fine-grained enough for each principal to assign unique timestamps to all important events occurring in that process.

Though physical clock synchronization is not a reliable assumption in all contexts, we expect that for many useful purposes, sufficient synchronization can be achieved with a protocol like NTP. Section 4.7 explains how synchronized physical clocks can be replaced with version matrices at the cost of a quadratic storage requirement. For the purpose of implementing continuous consistency in Section 5, timestamps from conventional scalar logical clocks can be used in place of physical clocks, which is how Yu and Vahdat implement TSAE as part of their TACT framework [42]. The real-time consistency metric discussed in 5.7 requires each process to keep a physical clock to measure elapsed time, though synchronization does not appear to be strictly required here either, assuming each clock runs at a consistent rate. We assume synchronization here to follow Golding's presentation, highlighting where the assumption is used.

4.2 Message Log

Each process P manages a *message log* containing all messages sent or received by P. We denote P's message log by Log^P. It grows in linear order, as a stack, as new messages are sent and received. Each message in Log^P is tagged with the identifier of the process that originally created it, with the subset of messages in Log^P that originated at Q denoted by $\mathsf{Log}^P[Q]$. Thus, $\mathsf{Log}^P[P]$ represents the set of messages created by P. The set of global messages, $\mathsf{Messages}$, is defined as the (disjoint) union of messages originating from every process, which by mild abuse of notation we are treating as sets:

$$\mathsf{Messages} \equiv \bigcup_{X \in \mathcal{P}} \mathsf{Log}^X[X].$$

The application at P broadcasts a message m using TSAE by writing m into Log^P. We call this event the *submission* of m and say that m originates at P. At the time of submission, the message is tagged with a pair $(P, \mathsf{clock}(P, t))$ containing P's identifier and current clock value. The timestamp of m is denoted by $\mathsf{time}(m)$.

Submitted messages are not sent to other processes right away, but propagate by a straightforward protocol. Periodically, P contacts some other process Q and the pair conduct an *anti-entropy session*. First, P and Q exchange summary vectors (defined below). P uses Q's summary vector to quickly calculate which messages in Log^P are not already known to Q and then transmits these messages to Q. Symmetrically, Q forwards messages in Log^Q that P has not seen, resulting in Log^P and Log^Q containing the same messages afterwards, though typically not in the same order.[6]

TSAE provides reliable eventual delivery, meaning every message is eventually received by every process, as long as no process fails and no partition indefinitely separates any two nodes. In particular, if processes stop originating new messages (so $\mathsf{Messages}$ remains fixed), if processes remain able to communicate, then Log^P is guaranteed to eventually converge to $\mathsf{Messages}$. To achieve this, it is enough that each pair of processes engage in anti-entropy periodically. This can actually be relaxed by transitivity: it is enough that they periodically engage in anti-entropy with a mutual partner, or with two other processes that engage in anti-entropy with each other, and so on. TSAE itself does not prescribe a particular policy regarding when and with whom to engage in anti-entropy, however. Consequently, it does not guarantee or estimate how far apart Log^P is from Log^Q, or from $\mathsf{Messages}$, at any moment. The framework in Section 5 prescribes a policy for TSAE based on conservative estimates of divergence in order to enforce these sorts of limits.

Note that the guarantee of eventual delivery only ensures that Log^P converges to $\mathsf{Messages}$ as a *set*. However, in general, we think of Log^P as a linear structure, with messages entered into Log^P in the order they were received by P. Thus, when Log^P and Log^Q eventually converge, it is more correct to say they will be permutations of each other rather than equal. For applications that require messages to be delivered in a particular order, like a total order, an additional ordering component is used, as in 4.5.

Example 4.1. Figure 21 shows a message log for containing six messages from three processes A, B, and C. For readability it is convenient to depict logs whose

[6] In a multithreaded environment where P and Q can engage in multiple anti-entropy sessions with different partners at once, the two sides may end in different states. For instance, P may learn about new messages in an anti-entropy session with R during its session with Q. Later, we also consider one-sided anti-entropy sessions that only push or pull messages.

contents are disaggregated by sender (Figure 21b). Disaggregation has the downside of obscuring the fact that message logs with the same contents may be arranged in different orders, as the reader should keep in mind.

4.3 Summary Vector

Besides Log^P, the message propagation component maintains a *version vector* $\mathsf{summary}^P$ whose role is to quickly communicate to other processes which messages P has already received. This structure is very similar to a vector clock (see Section 3.3.2) with a few differences. First, the version of TSAE presented here happens to store physical, rather than logical, clock values. A more essential distinction is that while a vector clock tracks causality between events and increments the local clock with each event, a version vector tracks the history of updates to replicated data. The value $\mathsf{summary}^P[P]$ tracks the set of messages sent by P and is incremented when P sends a new message, but not when P receives a message from another process. As with vector clocks, $\mathsf{summary}^P[Q]$ can be thought of as P's lower bound view of $\mathsf{summary}^Q[Q]$.

When P submits a message to its own log, $\mathsf{summary}^P[P]$ is advanced to $\mathsf{clock}(P, t)$ and the message is timestamped with this value. At some point in the future, P contacts some other process Q to initiate an anti-entropy session. We conceptualize this process as happening in three phases (setup, message exchange, and conclusion):

1. The value $\mathsf{summary}^P[P]$ is advanced to $\mathsf{clock}(P, t)$. Symmetrically, $\mathsf{summary}^Q[Q]$ is advanced to $\mathsf{clock}(Q, t)$. The partners exchange summary vectors.

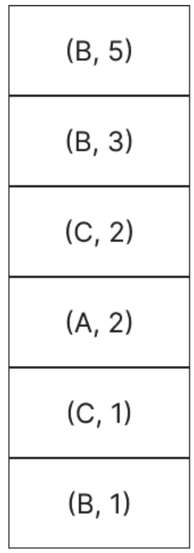
2. For each process $X \in \mathcal{P}$, P sends to Q the set of messages in $\mathsf{Log}^P[X]$ with timestamps greater than $\mathsf{summary}^Q[X]$, if any. Likewise for each X it receives all messages from $\mathsf{Log}^Q[X]$ with timestamps greater than $\mathsf{summary}^P[X]$ and adds these to $\mathsf{Log}^P[X]$. The partners exchange signals to indicate when they are finished sending and receiving updates.

3. Much like a vector clock, $\mathsf{summary}^P$ is updated to the pointwise maximum of its current value and the value of $\mathsf{summary}^Q$ received from Q. Likewise Q updates $\mathsf{summary}^Q$.

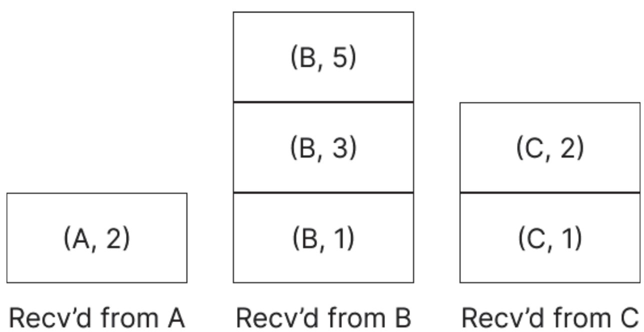
The message log satisfies an invariant, termed the coverage property, formalizing the role of $\mathsf{summary}^P$ as a summary of the contents of P's message log.

> **Coverage Property**: For all Q, P has received all messages originating at Q whose timestamps are less than the Q^{th} entry in P's summary timestamp vector:
>
> $$\{m \in \mathsf{Log}^Q[Q] \mid \mathsf{time}(m) \leq \mathsf{summary}^P[Q]\} \subseteq \mathsf{Log}^P[Q].$$

The previous subset relation is "morally" an equality—messages in $\mathsf{Log}^P[Q]$ but not the subset are said to be "early"—but only the subset property is required for



(a) A message log with entries accumulating bottom-to-top in linear order



(b) The same log shown with messages disaggregated by sender

Figure 21: A message log displayed linearly and disaggregated

correctness.[7] Thus, besides forming a lower bound of $\mathsf{summary}^{Q}[Q]$, $\mathsf{summary}^{P}[Q]$ can be thought of as an *upper* bound of the originating time, measured by Q's clock, of the last message P received from Q. The coverage property implies that $\mathsf{summary}^{P}$ provides complete information about the (non-early) contents of Log^{P}.

The reader should convince themselves that the following inequalities hold at all times for all P and Q (including when $P = Q$):

$$\max_{m \in \mathsf{Log}^{P}[Q]} (\mathsf{time}(m)) \leq \mathsf{summary}^{P}[Q] \leq \mathsf{summary}^{Q}[Q] \leq \mathsf{clock}(Q, t).$$

Example 4.2. Figures 22—28 depict the evolution of TSAE executing across three distributed processes A, B, and C. The figures depict the following chain of events beginning at $t = 3$. Note that by $t = 9$, A learns of writes submitted with C without performing direct communication with C. The figures also depict acknowledgment vectors, shown in red, and what we later term the commit line, shown underlined. These are explained below.

Time	**Action**
$t = 1$	A submits a write
$t = 2$	B and C submit writes
$t = 3$	B submits a write
$t = 4$	A and B begin anti-entropy and swap summary vectors
$t = 5$	B submits a new write
$t = 6$	A and B finish anti-entropy, C submits a write
$t = 7$	B and C begin anti-entropy, A submits a write
$t = 8$	B and C finish anti-entropy, A and B begin anti-entropy
$t = 9$	A and B finish anti-entropy

[7] Section 5.4.3 of Golding's thesis describes how a version of TSAE combined with an unreliable network-level multicast for optimization purposes can add messages to $\mathsf{Log}^{P}[Q]$ early.

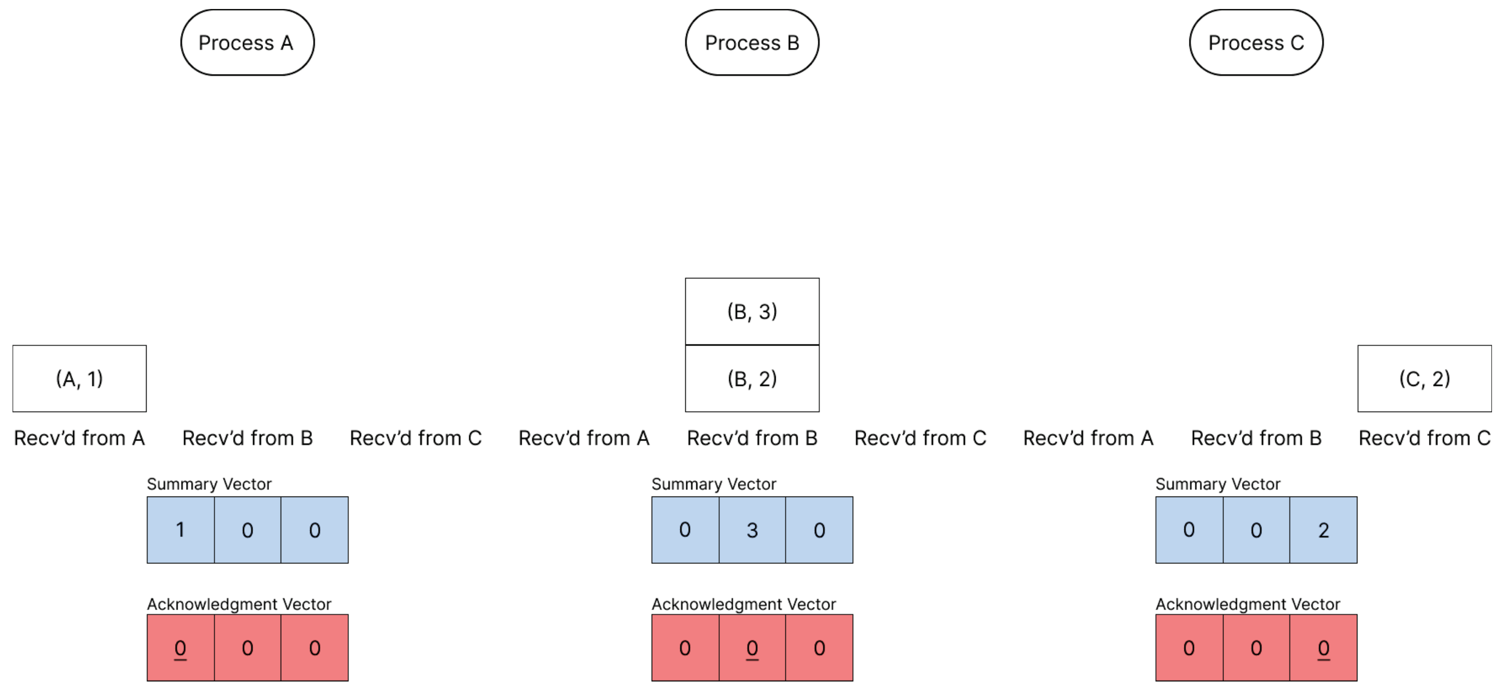


Figure 22: TSAE at time $t = 3$. Each process has submitted a write request (B has submitted two), but no process has notified any other process yet of the updates. This might be because they have not had the opportunity to exchange updates at this moment.

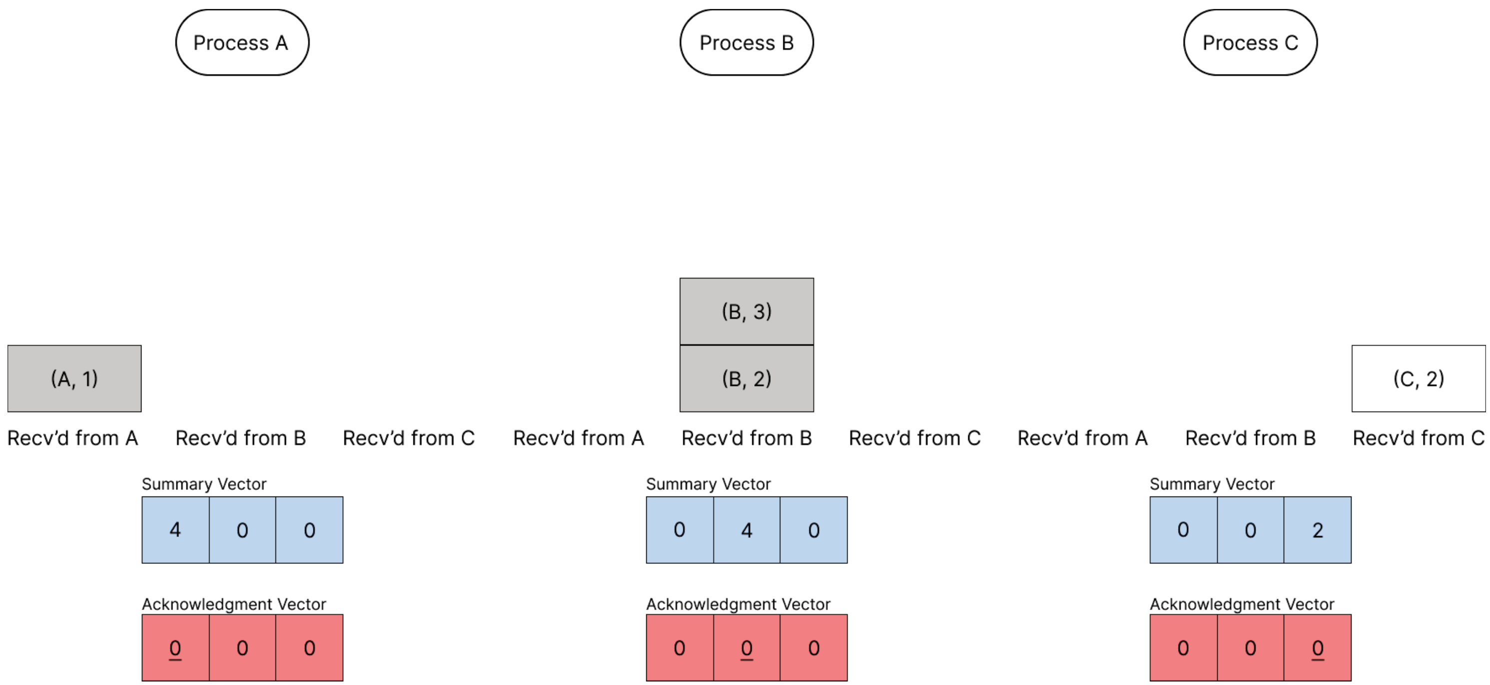


Figure 23: TSAE at time $t = 4$. A advances $\mathsf{summary}^A[A]$ to $\mathsf{clock}(A, t) = 4$ and likewise for B. After exchanging summary vectors, the participants decide to exchange the shaded messages.

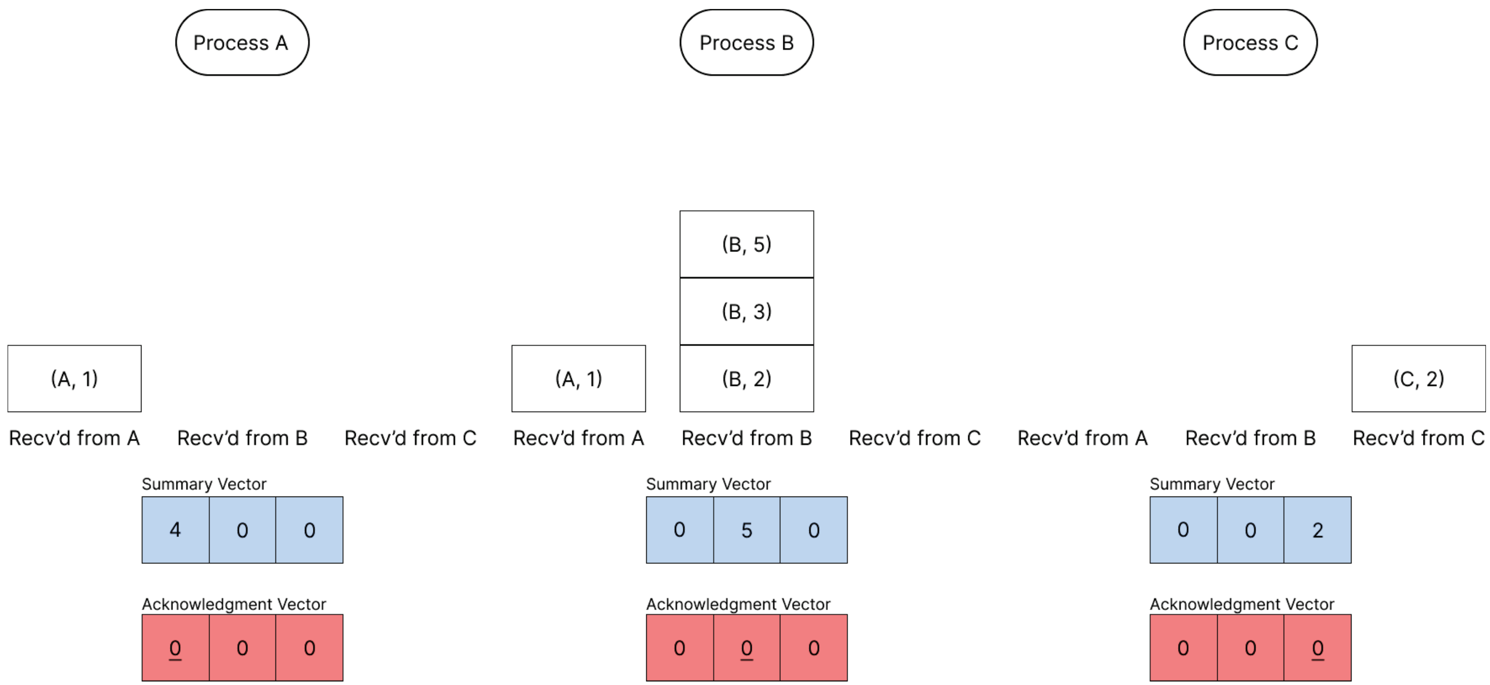


Figure 24: TSAE at time $t = 5$. B has submitted a message with timestamp $t = 5$ while A and B are still engaged in an anti-entropy session in the background.

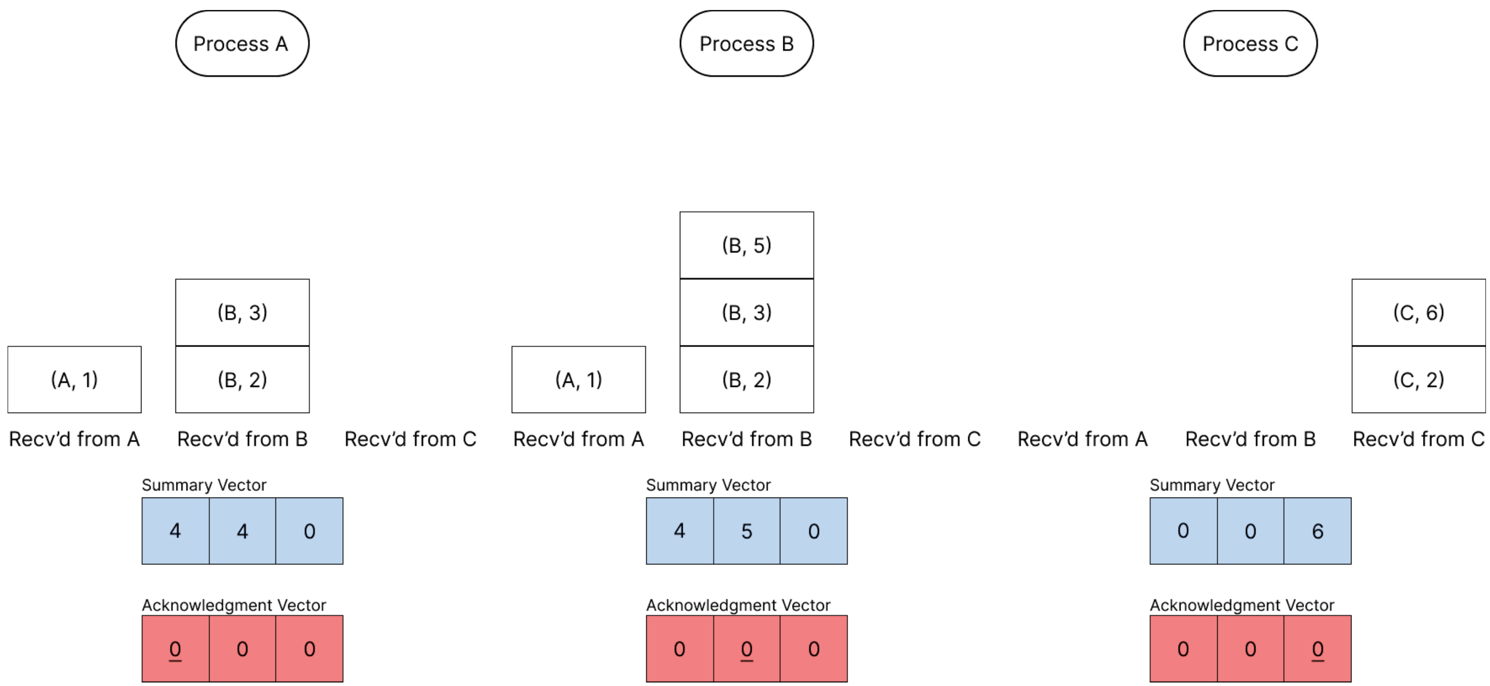


Figure 25: TSAE at time $t = 6$. A and B have finished their session and updated their summary vectors. Neither A nor B can update their commit line past 0 because they both contain $\mathsf{summary}[C] = 0$, indicating they have not seen any messages from C. C submits a message with timestamp $t = 6$.

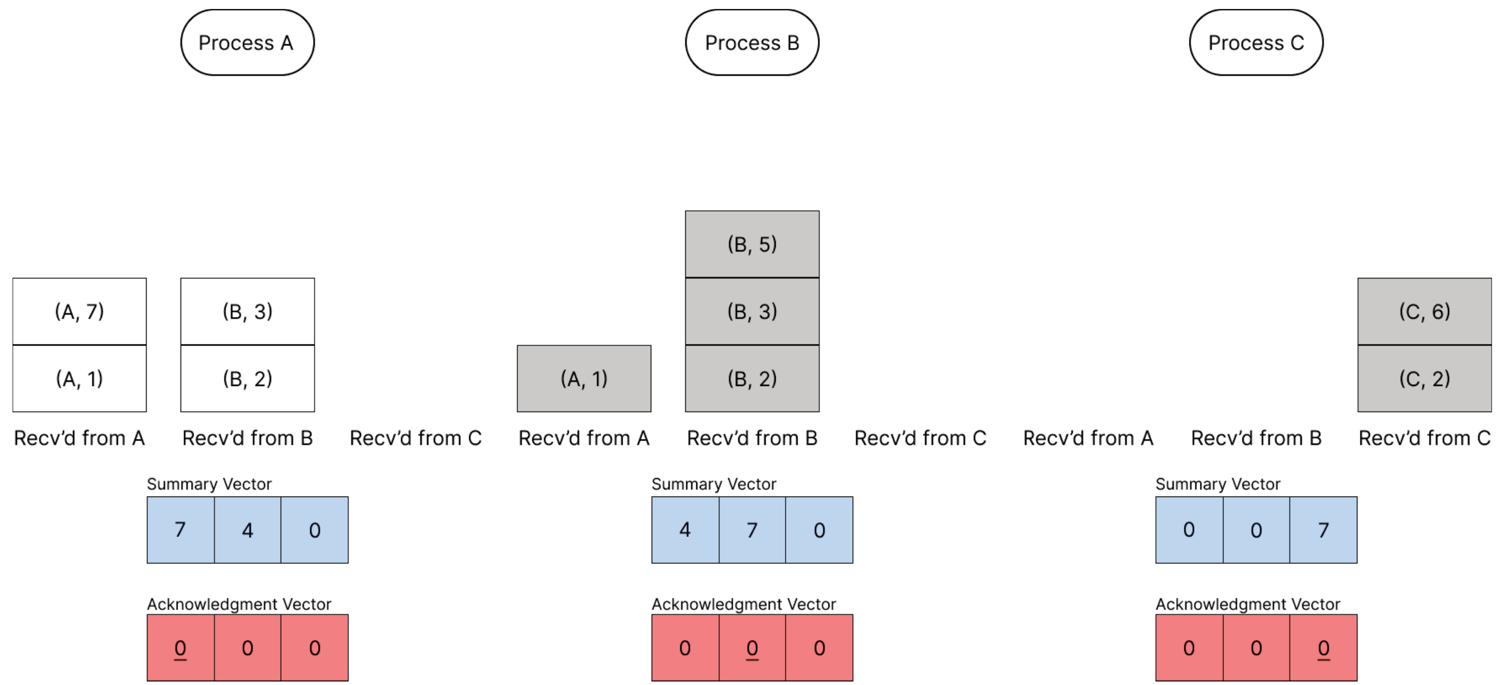


Figure 26: TSAE at time $t = 7$. B and C update and exchange summary vectors before deciding to exchange the shaded messages. A submits a message with timestamp $t = 7$.

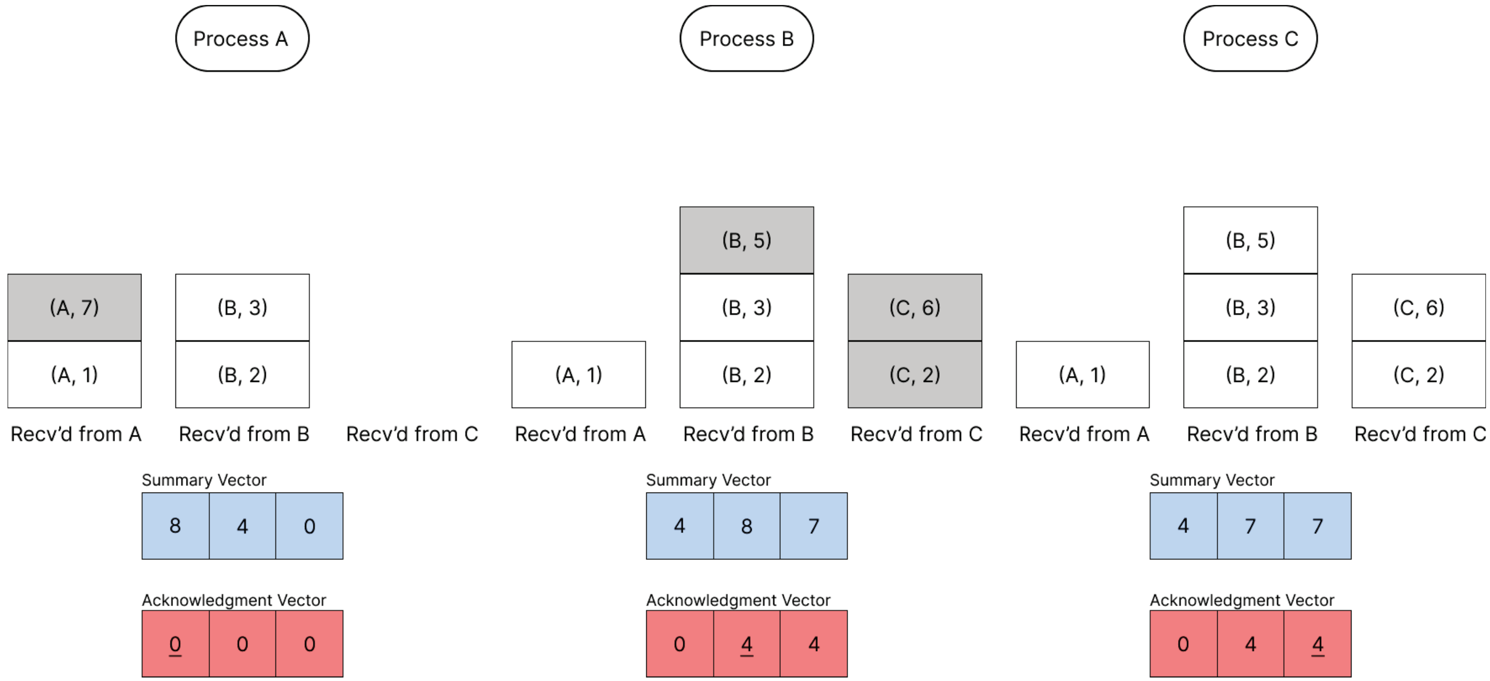


Figure 27: TSAE at time $t = 8$. B and C finish their anti-entropy session. Both sides can update their commit line to 4, since they have seen all messages m such that $\mathsf{time}(m) \leq 4$. A and B begin anti-entropy, exchanging updated summary vectors and exchanging the shaded boxes.

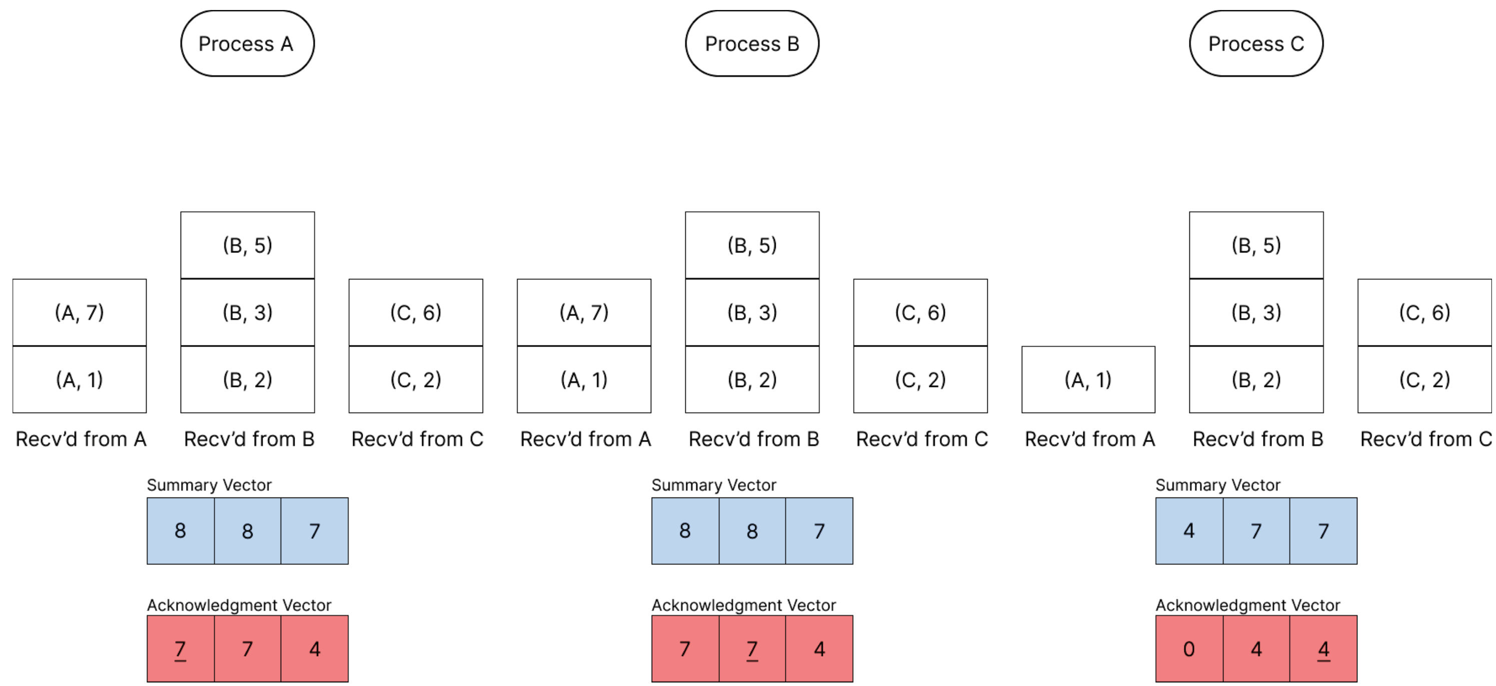


Figure 28: TSAE at time $t = 9$. A and B finish their anti-entropy session and have both seen all messages with timestamps less than or equal to $t = 7$. B and C can remove all messages with timestamps less than or equal to 4 from their logs, since they both have minimum entries $\mathsf{ack}\,[C] = 4$. C has received the same messages as A and B, but has not witnessed A acknowledging them.

4.4 Acknowledgment Vector

P's summary vector enables another process to quickly learn which messages P has seen. Additionally, P must keep tabs on which messages *other* processes have seen. This information is critical for the message ordering and log recycling components of TSAE. Golding presents two ways to maintain this information, which make different efficiency tradeoffs.

Arguably the simplest approach is for P to maintain a local (lower bound) copy, called P's *view*, of the summary vector of every other process. During anti-entropy sessions, P exchanges views with its partner alongside its $\mathsf{summary}^P$, taking their pointwise maximums afterwards. This leads to the idea of a version matrix, but it has the downside of a per-process space requirement that is quadratic in the size of the process group. This approach is considered further in 4.7.

The other mechanism presented by Golding requires each process P to maintain an *acknowledgment vector* ack^P. The basic idea is to coarsely summarize P's knowledge of other processes not with its summary vector, but the minimal element in this vector, a scalar, which is stored in $\mathsf{ack}^P[P]$. For reasons explained in Section 5, this value is called P's *commit line*. Periodically, P's commit line is updated to the minimal timestamp in its summary vector, $\min_{X\in\mathcal{P}}\left(\mathsf{summary}^P[X]\right)$. For correctness, the invariant required of $\mathsf{ack}^P[P]$ is that it is always a lower bound of this minimum.

> **Acknowledgment Property**: P's commit line is less than or equal to the minimum value in P's summary vector.[8]
>
> $$\mathsf{ack}^P[P] \leq \min_{X\in\mathcal{P}}\left(\mathsf{summary}^P[X]\right)$$

Acknowledgment vectors are updated during anti-entropy sessions much like vector clocks and version vectors.

1. At the beginning of the session, $\mathsf{ack}^P[P]$ is updated to $\min_{X\in\mathcal{P}}\left(\mathsf{summary}^P[X]\right)$. Symmetrically for Q. P and Q exchange acknowledgment vectors alongside their summary vectors.

2. At the end of the session, P sets ack^P and to the pointwise maximum of its current value and the value of ack^Q received from Q. Q updates ack^Q symmetrically.

Note that $\mathsf{ack}^P[Q] \leq \mathsf{ack}^Q[Q]$ at all times for all P and Q.

Slightly different from Golding's presentation of the protocol, the figures discussed in Example 4.2 also demonstrate an optimization that takes advantage of the fact that summary vectors are updated at the end of anti-entropy sessions. After updating these vectors to their pointwise maximum, we immediately recompute $\mathsf{ack}^P[P]$, setting it to $\min_{X\in\mathcal{P}}\left(\mathsf{summary}^P[X]\right)$. Furthermore, $\mathsf{ack}^P[Q]$ is set to this value. Symmetrically, Q updates $\mathsf{ack}^Q[Q]$ and $\mathsf{ack}^Q[P]$. Note that this preserves the invariant that $\mathsf{ack}^P[Q]$

[8] The inequality here can be thought of as morally an equality. It may be a strict inequality while P is updating $\mathsf{summary}^P$ before recomputing its minimum. Recall P may be multi-threaded with multiple anti-entropy sessions affecting $\mathsf{summary}^P$ at the same time.

and $\mathsf{ack}^Q[P]$ are lower bounds of $\mathsf{ack}^Q[Q]$ and $\mathsf{ack}^P[P]$, respectively, provided the implementation ensures P increments $\mathsf{ack}^P[Q]$ after confirming Q has received all messages from P and vice versa. This optimization advances acknowledgment vectors (and later, version matrices) more often than Golding's presentation, which allows the ordering and purging mechanisms (explained below) to progress more quickly.

The following two lemmas explain the utility of ack^P as a way of estimating global state.

Lemma 4.3. *P has received all messages (from any sender) with timestamps less than or equal to its commit line $\mathsf{ack}^P[P]$.*

Proof. Let message m originate at Q with timestamp $\mathsf{time}(m) \le \mathsf{ack}^P[P]$. Then

$$\mathsf{time}(m) \le \mathsf{ack}^P[P] \le \min_{X \in \mathcal{P}} \left(\mathsf{summary}^P[X]\right) \le \mathsf{summary}^P[Q].$$

The coverage property implies $m \in \mathsf{Log}^P$. □

Lemma 4.4. *For all messages m, if $\mathsf{time}(m) \le \mathsf{ack}^P[Q]$, then Q has received m.*

Proof. Now $\mathsf{time}(m) \le \mathsf{ack}^P[Q] \le \mathsf{ack}^Q[Q]$. By Lemma 4.3, Q has received m. □

When a message m in the write log satisfies $\mathsf{time}(m) \le \mathsf{ack}^P[P]$, P is said to have *acknowledged* m. In light of Lemma 4.3, all acknowledged messages have been received. The converse does not hold, since a received message will not be acknowledged until each entry in $\mathsf{summary}^P$, not just the entry of its sender, is greater than $\mathsf{time}(m)$. We now explain how acknowledged messages can be delivered to the application and ultimately purged from Log^P, before considering the matrix-based alternative to acknowledgment vectors.

4.5 Message Ordering

TSAE guarantees that messages will eventually be received by all other processes, but ordering of these messages in different processes' logs may vary. In some applications this is acceptable, but other applications require more control over delivery order. Recall from Section 3.4 that to enforce ordering guarantees, a distinction is made between message receipt and delivery. A message ordering layer buffers incoming messages upon receipt, giving time for slower messages to catch up to faster ones, before delivering them to the application in a predictable order. Below, we describe a simple implementation of totally ordered delivery for TSAE. Golding also describes straightforward mechanisms to enforce causal and total-causal order in Section 5.5 in his dissertation.

The message ordering component of TSAE periodically inspects Log^P and delivers messages to P when ready. To enforce total order, this component delivers all messages whose timestamp is less than or equal to P's commit line. That is, the set of messages satisfying

$$\mathsf{time}(m) \le \mathsf{ack}^P[P]. \tag{1}$$

These messages are delivered to P in order of their timestamps, using the identifiers of their senders to resolve ties. We refer to the combination of TSAE with total ordering as TSAE+TO. This protocol has the following eventual convergence property.

Lemma 4.5. *TSAE+TO eventually delivers every message to every process in the same order.*

Proof. Assuming periodic anti-entropy sessions, the protocol in Section 4.2 ensures that each message is eventually received by P and each value of the form $\mathsf{summary}^P[X]$ eventually increases. Thus, the minimum entry in $\mathsf{summary}^P$, and thus $\mathsf{ack}^P[P]$, will eventually increase beyond $\mathsf{time}(m)$, so each m will be delivered as long as the ordering component runs periodically. The fact that messages are delivered in the same order everywhere follows from Lemma 4.3, since P has seen all messages whose timestamp is less than $\mathsf{ack}^P[P]$, and therefore the stream of deliverable messages does not have any "gaps" in the total order. □

The above mechanism for enforcing a total order dates back to Lamport's introduction of scalar clocks [22], which Lamport used to implement a replicated state machine (RSM), a conceptual device whose status is uniquely determined by a history of transitions applied to a starting state. Two conditions ensure replicas converge to the same state. First, the *contents* of their histories agree (they have seen all the same transitions), and second, their *orders* agree (they have applied transitions in the same order). Using TSAE to announce transitions ensures the first condition, since every process will eventually learn about each transition. Combining TSAE with a TO delivery mechanism ensures the second condition, if transitions are applied when announcements are delivered. This provides a form of *weak* consistency—it ensures all replicas eventually reach the same state, which makes it inherently fault-tolerant, though it does not provide guarantees about the observed inconsistency of replicas in the meantime. We describe using TSAE+TO for database replication in Section 5, incorporating the "conit" framework of Yu and Vahdat [42] to bound observed inconsistency.

Golding's assumption of loose clock synchronization is driven by a practical need to ensure messages are delivered (and also purged, described below) in a timely fashion instead of just eventually. For example, if Q has an exceptionally slow clock compared to other processes, then $\mathsf{summary}^P[Q]$ will remain the minimum element in $\mathsf{summary}^P$ for a long time. During this time, only messages from Q will be delivered to P, as all other messages would have timestamps greater than Q's clock. Section 4.7 explains one way, also introduced by Golding, that the assumption of synchronization can be partially mitigated. The TSAE-based protocol in Section 5 does not require synchronized clocks except to bound real-time staleness.

4.6 Message Purging

A message simply cannot be removed from the write log after being delivered to P, because some of the messages in Log^P may be propagated to new recipients during anti-entropy in the future.[9] However, it is untenable to let the message log grow without limit. Thus, a separate log recycling process can be used to remove old entries when they are no longer required.

[9] This implies the ordering component must perform bookkeeping to remember which messages in the log have already been delivered.

There are two requirements for a message to be safe to delete. First, it clearly must have been delivered to P already. Additionally, it must have been received by all other processes—otherwise it might be one of the messages P should send in a future anti-entropy session. P will know a message m has been received by all other processes when its timestamp is less than or equal to P's *purge line*, defined as the minimum entry in the acknowledgment vector:

$$\mathsf{time}\,(m) \leq \min_{X \in \mathcal{P}} \left(\mathsf{ack}^P\,[X]\right) \tag{2}$$

The safety of this deletion procedure is proven by the following lemma.

Lemma 4.6 (Log purging)**.** *P can safely discard all messages in Log^P with timestamps less than or equal to its purge line, after they have been delivered.*

Proof. Let message $m \in \mathsf{Log}^P$ originate at R with timestamp less than P's purge line. Now the following inequalities hold for all Q:

$$\mathsf{time}\,(m) \leq \min_{X \in \mathcal{P}} \left(\mathsf{ack}^P\,[X]\right) \leq \mathsf{ack}^P\,[Q]\,.$$

Therefore, by Lemma 4.4, m has been delivered to Q. Since Q is arbitrary, m has been received everywhere (where it will eventually be delivered as well) and can be purged from Log^P to reclaim storage space. □

Because every message is eventually delivered and acknowledged by each process, P will eventually be able to remove each message from its log. It is still possible for Log^P to grow without bound if the rate of message arrival exceeds the speed at which they are purged. This might occur during periods of heavy usage or during a network partition, as P will eventually become unable to advance $\mathsf{ack}^P\,[Q]$ further if Q is on the other side of a partition. In this situation, further mechanisms can be incorporated (and in practice, they often are) for P to decide that Q is no longer a participant, and resort to some type of failure mode, with the possibility of later reincorporating Q into the group. Of course, such mechanisms add to the complexity and subtlety of distributed systems protocols, and they are outside this scope of this document. In any case, the storage requirements of the message log in a particular use case and environment should be measured empirically as part of an application optimization strategy.

4.7 TSAE using Version Matrices

Using a single value, $\min_{X \in \mathcal{P}} \left(\mathsf{summary}^P\,[X]\right)$, to concisely estimate which messages P has received has certain drawbacks. If clocks are not approximately synchronized, the clock value at one process may greatly exceed that of another. If Q has a very fast clock, this value will remain less than $\mathsf{summary}^P\,[Q]$ for a long time and messages from Q will not be purged. If Q has a slow clock, only messages from Q will be delivered and purged.

Rather than the minimum entry, the entire summary vector offers a more precise measure of Log^P. For P itself, that means tracking what other process know about by keeping a copy of $\mathsf{summary}^Q$ for each Q in the system. Thus, we do away with

ack^P and track what other processes know using a vector-of-vectors that we will denote matrix. Since $\mathsf{summary}$ is a version vector, then matrix can be called a version matrix.

With this implementation strategy, $\mathsf{matrix}^P[P]$ stores P's summary vector, while $\mathsf{matrix}^P[Q]$ represents P's lower bound estimate of Q's summary vector. Thus, $\mathsf{matrix}^P[Q][R]$ is P's lower bound estimate of the upper bound of the timestamp of any message Q has received from R. Matrices are exchanged during anti-entropy sessions just as before, with both sides taking the pointwise maximum after.

We additionally apply an optimization, similar to the one for ack^P, of recomputing matrix^P after confirming Q has successfully received all messages during anti-entropy. Namely, $\mathsf{matrix}^P[Q]$ is advanced to the pointwise maximum of the summary vectors $\mathsf{matrix}^P[P]$ and $\mathsf{matrix}^Q[Q]$ exchanged at the beginning of the anti-entropy session. This increases P's estimated knowledge of Q to reflect any new messages P just pushed during anti-entropy. (The previous value reflected only the messages Q knew about when the session was initiated.) Q updates $\mathsf{matrix}^Q[P]$ symmetrically. For purposes of continuous consistency in Section 5, this subroutine implements the "view advance" mechanism used to bound numerical error.

For message ordering and log recycling, P applies the following (conservative) policies:

- A message with timestamp m is ready to be delivered to P by a total ordering component when the following analogue of (1) holds:

$$\mathsf{time}\,(m) \leq \min_{X \in \mathcal{P}} \left(\mathsf{matrix}^P[P][X]\right) \tag{3}$$

- A message m originating at R has been received by Q when the following condition holds:

$$\mathsf{time}\,(m) \leq \mathsf{matrix}^P[Q][R] \tag{4}$$

- A message m originating at R, after being delivered to P, can be purged from the log when the following analogue of (2) holds:

$$\mathsf{time}\,(m) \leq \min_{X \in \mathcal{P}} \mathsf{matrix}^P[X][R] \tag{5}$$

Note that the matrix-based version of TSAE does not necessarily solve the delivery problem if clocks are unsynchronized: if physical timestamps are used to totally order all messages in the system, messages from a host with a slow clock will be delivered before any other messages.

Of course, matrices require $\Theta(N^2)$ storage at each site for a process group with N members, which quickly becomes untenable for systems where N is on the order of 1,000 or greater.

5 Continuous Consistency

We now turn our attention back to the topic of distributed memory consistency, first introduced in Section 3.5. We will unite the main themes in this document by explaining the ideas behind *continuous* consistency. We particularly focus on the conit (short for "consistency unit") model proposed by Yu and Vahdat in a series of papers [40, 44, 43, 41, 42]. Accompanying their work is a prototype implementation called TACT (Tunable Availability and Consistency Toolkit), a Java-powered middleware that uses timestamped anti-entropy to maintain a configurable level of consistency between replicas of a shared database.[10]

The conit approach to consistency is motivated by the observation that many real-world applications can tolerate a certain amount of divergence among replicas in exchange for greater application performance. However, they cannot tolerate too much inconsistency: if the divergence exceeds a certain threshold, the data becomes too unreliable for its intended use. For these kinds of applications, rather than viewing consistency as a yes-or-no condition, as with linearizability and sequential consistency, it is desirable to have a consistency model that numerically quantifies levels of inconsistency and provides controls to "tune" the application by tightening or relaxing the allowed amount of divergence between replicas. This makes the consistency/availability tradeoff into not merely an abstract concept but a tangible one, allowing applications to choose where they are located on a spectrum between total consistency (i.e. linearizability) and total availability (i.e. never blocking to enforce consistency requirements).

Mechanisms to exchange consistency for availability, or vice versa, provide advantages for distributed shared memory frameworks deployed across disruption-prone networks. In these contexts, message-passing latencies virtually guarantee that some amount of global inconsistency is inevitable. However, the safety-related nature of the applications we have in mind makes this inconsistency important to measure quantitatively and then limit. Of course, imposing tighter limits on inconsistency eventually comes at the cost of system performance. The advantage of the conit model is that the precise tradeoff can be configured according to policies set by the application, rather than being determined directly by the consistency model. Indeed, it is possible to adjust these policies at runtime, perhaps in response to factors like application workload and network capacity, which might provide the precise sort of behavioral continuity foreshadowed in Section 2.1.

5.1 System Model

The system consists of processes in a group $\mathcal{P}$ that replicate a shared database or, more generally, any kind of replicated state machine. We shall assume that every process maintains a complete replica of the database. For simplicity, we assume that the set of participating processes is fixed and known to each member, though one can imagine extending the framework to support dynamic group membership following Golding [15].

Each process begins with their replica in the same initial starting state, denoted D_{init}. For each participating process $P \in \mathcal{P}$, we denote the current state of P's replica with D^P. D^P changes as P learns about updates to the data, which can come from two sources: a request submitted to P by a client of the application, or a

[10] The source code to the TACT prototype, built in the early 2000's, was evidently not made publicly available.

notification about a request submitted to some other process by one of its clients. For a request submitted to P directly by a client, D^P is said to be the *originating replica*. For all other requests, D^P is said to be a *remote replica*.

As in Section 3.5, we imagine requests divided into two types: read requests and write requests. We call either type of request a data *access*. Unlike in Section 3.5, we allow for write accesses that are more complex than simple instructions that set the value of a logical memory location. Instead, here we are viewing a write as more like an entire *database transaction*—a potentially complex set of steps that can read several data items, modify them, and might also return a value to the user. In fact, read accesses should be thought of as merely special cases of write accesses that do not modify any data. Later, we will describe how the concepts of "read error" and "order error" measure the significance of an update when applied to a database replica—then, a read request can be thought of as an update that measures zero with respect to both of these metrics, because it does not change the database state.

It is important to note that a write access can encode application-level logic. For example, it may read a value in the database and make a business decision whether to increment it. The resulting state after applying write action w to a database in state D is denoted $D+w$. Because reads are just accesses that happen not to modify the underlying database, we have the equation $D + r = D$ if r is a read. In many real-world data models, writes are not commutative, meaning in general we have the following inequality:

$$D + w + w' \neq D + w' + w \tag{6}$$

For such applications, consistency requires writes to be applied in the same order everywhere. Therefore the conit framework ensures that processes can enforce a global total order among writes. We shall explain later how it also provides a mechanism for individual replicas to optimistically deviate from the total order within controllable bounds, which can improve application performance.

Recall that a *history* is an ordered sequence of read and write accesses. If H is any history, for each access $a \in H$ in the history, the *prefix* history of a, denoted $\mathsf{PH}(a)$, is the sequence of accesses in H that precede a. The *local history* of a process P, denoted H^P, is the sequence of accesses P has applied to the initial state D_{init} since the start of the application. Let $D + H$ denote the result of applying each of the accesses in H to D in history order. Thus, the database state at process P at any moment, written D^P, is governed by the following equation:

$$D^P = D_{\text{init}} + H^P. \tag{7}$$

The effect of executing a read or write access a against the replica at P is entirely determined by the local history of P at the time a is submitted. Because, in general, accesses can encode arbitrarily complex application-level logic, the state of D^P observed by a can strongly influence how a is executed and what value it returns to the user.

Definition 5.1. The *observed prefix history* $\mathsf{PH}_{\text{obs}}(a)$ of an access a originating at P is the value of H^P immediately before a is applied to D^P. The *observed state*, denoted $D_{\text{obs}}(a)$, is the state of D^P at the time a is applied, which gives the equation

$$D_{\text{obs}}(a) = D_{\text{init}} + \mathsf{PH}_{\text{obs}}(a) \tag{8}$$

Intuitively, the goal of the conit framework is to ensure that each access observes a value of D^P whose divergence from its correct or "ideal" value is limited by an upper bound. The ideal value of D^P is the state it would be in if it has received all updates and applied them locally in the correct order. In the conit framework, there are two sources of error that can cause a replica to diverge from its ideal value:

Unseen updates State transitions originating at other processes that have not been applied to D^P because P has not been notified of them yet

Our of order updates State transitions P has applied locally in the wrong order, leading it down an "errant path" beginning with first update that disagrees with the global order

Unseen updates occur when a write access is submitted by a client of some other process Q and P is not notified right away, making P's replica stale. The conit framework presents two metrics with respect to which we can limit unseen updates. The first, numerical error, limits the total "weight" of unseen updates, where weight is a conceptual measure of how significant an update is with respect to the data model. Numerical error is the subject of Section 5.5. The second, real-time staleness, limits the maximum amount of time that may pass before a process is made aware of a new update. Real-time staleness is the subject of Section 5.7.

Out of order updates can occur because when P receives an update (or a batch of updates), whether submitted by a client locally or forwarded to P by another process, P generally applies the update to its own replica immediately, *bypassing* the total-order delivery mechanism. This may improve performance because the application might otherwise have to block while the ordering mechanism determines its place in the total order. This is an I/O-bound operation, as described in Section 5.6, since it generally requires pulling information from other processes over the network to advance P's commit line.

Recent updates that have been applied without running the total-order delivery mechanism are said to be *tentative*, while the set of updates known to have been applied in the correct order (always a prefix of H^P) are *committed*. Some time later, the total-order delivery mechanism may determine that a subset (always a suffix) of the tentative updates were applied in the wrong order. The framework provides a mechanism to rollback these incorrect updates and reapply them in the correct order. "Out-of-order" error sets a limit on the weight of tentative updates before the total-order mechanism must be invoked. The "order weight" of an update is separate from the numerical weight, and may be thought of as the potential cost associated with rolling back an update later discovered to have been applied in the wrong order. Out-of-order error and the commitment process are the subject of Section 5.6.

Consistency is measured at the granularity of conits, or consistency units, which are defined by the programmer. They are implemented simply by specifying the numerical and order weight of each update on a per-conit basis. A particular update may be deemed to change one conit significantly and another only slightly without affecting a third conit at all. Likewise, each process specifies its tolerable error, broken down by numerical error, order error, and real-time staleness, for each conit. The middleware is responsible for ensuring that each process' error bounds are enforced for each conit for each metric. If the error bounds are very tight, the system will spend more time blocking and less time responding to users. If the error bounds are very loose, users will observe a more responsive system, with relatively less assurance of the consistency of what they observe.

5.2 Write Propagation

We imagine process in $\mathcal{P}$ communicate over a disruption-prone network, perhaps a tactical mesh network deployed in the field. The details of this network are not salient to the discussion, except to emphasize that such a scenario does not lend itself to centralized solutions where a single replica is maintained that all processes access through a designated server.

The first requirement to implement the conit framework is a mechanism for reliable eventual totally-ordered delivery to propagate write accesses between the processes in $\mathcal{P}$—this is a general method for replicating a state machine. The implementation we describe is the one proposed by Yu and Vahdat, which is based on TSAE+TO, or timestamped anti-entropy with a totally ordered delivery component, defined in Section 4.5. Each process P maintains a write log Log^P that functions as the message log in TSAE. P also maintains the other data structures required for TSAE, whether in the form of summary and acknowledgment vectors (Sections 4.3 and 4.4) or a summary matrix (Section 4.7).

Writes originating at P are tagged with P's identifier and clock value (whether physical or logical) and entered into Log^P when they are applied to D^P. Remote write accesses added to Log^P during anti-entropy are applied to D^P in the order they are received. Thus, at all times Log^P contains the history of all updates applied to D^P and we have the following equation:

$$D^P = D_{\text{init}} + \mathsf{Log}^P \tag{9}$$

This follows from Equation (7) and the observation that $D + r = D$ if r represents a read access. Reads are handled locally by P without notifying remotes or entering the access into Log^P.

Using TSAE, P uses the summary vectors of remote replicas to determine which writes it may need to send them, and likewise sends its summary vector to others to receive updates from them. Messages are only ever sent in batches determined by comparing summary vectors, and properties like the coverage property (Section 4.3) are maintained as invariants.

One feature of the version of TSAE presented here is that we allow anti-entropy to be one-sided, where only one side learns about new updates and advances its summary vector. We describe one-sided sessions as push-based or pull-based depending on whether the sending or receiving side initiates the session:

Push based P initiates a session by requesting the summary vector of Q. P uses this information to decide which writes P is aware of that Q has not been informed about, pushing these updates to Q.

Pull based P initiates a session by sending its summary vector to a remote replica Q. Q uses this information to determine which writes to send to P.

Additionally, unlike in "pure" TSAE, the implementation we describe introduces a notion of compulsory anti-entropy sessions. Recall that the progress guarantee provided by TSAE is only that messages are delivered eventually, and so it does not place any requirement on how often anti-entropy sessions are scheduled except that they happen periodically. To bound the divergence between replicas at all times, the conit framework requires processes to become unavailable and engage in one-way anti-entropy sessions to increase consistency whenever responding to a user's request immediately would violate consistency requirements.

5.3 Ideal States

To define the correctness of the conit framework, we need a history to serve as a common reference point with respect to which each local history is measured. Intuitively, this history should represent one that might be observed if the middleware submitted all requests to a central database instead of maintaining separate replicas. This history should be "global" in the sense that it contains every access, which is equivalent to taking the union $\bigcup_{X \in \mathcal{P}} H^X$ of all local histories (more precisely, the union of their underlying sets—this notation is implicitly ignoring their ordered structure). Note that accesses are uniquely determined by the tag containing the name of their originating process, paired with the timestamp of that process at the time it was submitted.

Recall from Definition 3.21 that an access a externally precedes a' if a has returned to the caller before a' is invoked. Yu and Vahdat also define a notion of causality:

Definition 5.2 (Causal precedence of accesses)**.** Access *a causally precedes* a' if a was already in the local history of the originating replica when a' was submitted (and so could have influenced how a' executed).

The necessity of Definition 5.2 is seen by imagining that a and a' represent long-running isolated database transactions executed concurrently. In this case, one operation will logically take effect (and be entered into the write log) before the other, which is a causal relation, but the operations have no external order relation.

With the preceding definitions, we can define an idealized global history to serve as a reference point for measuring divergence.

Definition 5.3. A *ECG* (externally consistent, causally consistent, global) history contains every access in the system, arranged in some total order, $<$, that is consistent with both external and causal order among accesses.

$$H_{\text{global}} = \langle \bigcup_{X \in \mathcal{P}} H^X, < \rangle \tag{10}$$

We will refer to an ECG history as simply a "global" history—the fact that this history must respect external and causal order is implicit. A global history provides a notion of an ideal database state against which to measure each replica.

Definition 5.4. Relative to a global history H_{global}, the ideal database image is the state it would be in after applying all the updates in the global history in order. If H_{global} is given as a global history, the ideal database image is defined as follows:

$$D_{\text{ideal}} = D_{\text{init}} + H_{\text{global}} \tag{11}$$

Note that a global history is not uniquely determined by the set of local histories, because external and causal order do not uniquely determine a global total order. The correctness condition of the conit framework states that at all times, some global history exists such that each replica is within some error margin of the ideal database state for this history. For now we assume some global history H_{global} has been given. This is needed to define the metrics used to measure a replica's divergence from its ideal value.

Definition 5.5. The *ideal prefix history* $\mathsf{PH}_{\text{ideal}}(a)$ of an access a is defined as the prefix of a as it occurs in the global history H_{global}. Similarly, the *ideal state* of an access is defined as the state of the database at the time a is applied in the ideal history:

$$D_{\text{ideal}}\,(a) = D_{\text{init}} + \mathsf{PH}_{\text{ideal}}(a) \tag{12}$$

The conit framework seeks to ensure that each local history remains within some distance of the global history, with the difference consisting of a limited number of unseen or out-of-order updates. Roughly, the statement of correctness has the following form:

> "At any moment, there exists some ECG history H_{global} such that each access submitted to any process P observes a level of consistency within some margin ϵ of what it would observe in the idealized ECG history."

The preceding is roughly translated into the following informal equation for all accesses a:

$$|D_{\text{obs}}\,(a) - D_{\text{ideal}}\,(a)\,| < \epsilon \tag{13}$$

To interpret this equation, we would need a way to quantify the difference between two replicas of a database. We now turn our attention to this topic.

5.4 Measuring Divergence

At any moment, it is likely that no local replica perfectly matches the ideal database state. Intuitively, clients of process P can usefully operate with the application as long as the difference is bounded above by some application-defined margin of error. Thus, we must provide a framework to measure this margin of error.

5.4.1 Units of Consistency

A complicating factor of defining a general framework is that applications have unique data models. Note that we have not made any assumptions about the format of the backing data store. In the simplest case, the data may be a key-value store, such as a simple model that associates names like $x, y, z \ldots$ with numerical values. This sort of simple memory model was used in Section 3.7. In real-world applications, the database may be something like a relational database, a database optimized for use in Geographic Information Systems (GIS), or a bespoke data model uniquely designed for the application. It is not clear a priori how to design a general framework to measure divergence without making detailed assumptions about the nature of the data.

Indeed, it is not even clear what the fundamental *unit* of consistency is—it may be the case that two replicas totally agree on some parts of the data, but disagree on other parts. This intuition leads one to consider a more granular model of consistency than one that measures divergence at the level of whole replicas.

The conit framework strikes a balance between being general enough to work in applications with different data models and practical enough to efficiently enforce consistency with simple protocols. The question of what constitutes a unit of enforceable consistency is handled simply: the decision is entirely delegated to the programmer. In terms of how they are implemented, a "conit" is nothing but an identifier invented by the programmer. Somewhat arbitrarily, we refer to conits with names like F, F_1, F_2, and so on.

Rather than attempting to provide a general theory by which the middleware can define and measure the significance of an update to the database, the framework inverts responsibility: the application programmer is entirely responsible for informing the middleware how much change each update applies to each conit. These values are not enforced by the middleware, but simply accepted as ground truths as used as inputs for further decision-making.

5.4.2 Dimensions of Consistency

Each process P specifies its consistency requirements for conit F with a tuple of three values:

$$\langle \mathsf{NE}\left(P, F\right), \mathsf{OE}(P, F), \mathsf{RTE}(P, F)\rangle$$

These stand for Numerical Error, Order Error, and Real Time Error, respectively. Recall that the disagreement between D^P and its ideal value comes from two sources: messages that have not been seen, and tentative writes applied in an order differing from the final global order. Numerical error and real-time error (used interchangeably with staleness) are both forms of unseen error, and out of order error, naturally, bounds the number of messages applied out of order. We describe each error metric below, starting with numerical error.

Each write is associated a well-defined numerical weight defined for each conit, though this may be 0 for conits that the write does not modify. The numerical error of a replica is always measured at the level of weight applied to individual conits.

Definition 5.6. The *numerical weight* of a write w on a conit F is a real number $\mathsf{NumWeight}(w, F)$ supplied by the programmer when w is submitted to the replication middleware.

Intuitively, $\mathsf{NumWeight}(w, F)$ measures the change that w has on the data item(s) represented by F. Since the middleware cannot calculate this value itself, the programmer must make the decision, although in some cases there is an "obvious" choice.

Example 5.7. If the database is a key-value store of numerical values, then the programmer might define a conit, say F_x, for a key named x. A write update that increments x by 1 should be declared with $\mathsf{NumWeight}(w, F_x) = 1$. More complicated conit definitions are possible as well. The programmer might define a conit F_{sum} to correspond to the sum of keys x and y. In this case, a write that increments x by 3 and subtracts 1 from y should be specified with $\mathsf{NumWeight}(w, F_{\mathsf{sum}}) = 2$.

$\mathsf{NE}\,(P, F)$ sets a bound on the total numerical weight applied to F of writes that P is unaware of. Setting this value to 0 means P will be notified of every write access that affects F originating at any process before that write request is finished executing and returns to the client who submitted it. This process will be described in Section 5.5.

Separately from numerical weight, each write is associated with an order weight for each conit.

Definition 5.8. The *order weight* of a write w on a conit F is a non-negative real number $\mathsf{OrderWeight}(w, F)$ supplied by the programmer when w is submitted to the replication middleware.

The order weight of w on conit F is best thought of as a measure of liability, namely the penalty the application (or users of the application) must pay if w is applied out of order and has to be rolled back later. The precise definition and utility of measuring order error will be given in Section 5.6.

$\mathsf{OE}(P, F)$ bounds the maximum weight of updates that affect F that P might apply out of order—setting this value to 0 means all writes that affect F will be applied locally by P in their correct order relative to each other, where "correct" is defined with respect to some global history H_{global} whose existence is guaranteed by the conit framework.

How the middleware associates a write w with numerical and order weights is an implementation detail. The approach described by Yu and Vahdat [42] has the programmer invoke function calls of the form $\mathsf{AffectConit}(F, x, y)$ inside in the body of functions that modify the backing data store. Here, F is the conit, x is $\mathsf{NumWeight}(w, F)$, and y is $\mathsf{OrderWeight}(w, F)$.

The third metric used to measure consistency is real-time staleness, which bounds the real time that is allowed to elapse between when a write is originated and when some other process is notified of it. Because it is measured in terms of physical time, the programmer cannot specify the "real time weight" of a write on a conit.

$\mathsf{RTE}(P, F)$ bounds the real time staleness of P with respect to any write that affects F. Setting this value to 5 seconds, say, implies that P will always be aware of any write affecting F that finished executing more than 5 seconds ago. Setting either of $\mathsf{RTE}(P, F)$ or $\mathsf{NE}(P, F)$ to 0 would make the other effectively 0 as well, though the metrics are enforced with different mechanisms and would exhibit different behavior (both being correct in the sense of preserving the correctness properties, but with different application performance). Real-time staleness is described in Section 5.7.

5.4.3 Enforcing Error Bounds

Each read or write access submitted to P is responsible for informing the middleware about which conits the access depends on. These are the conits that logically represent data in D^P that might influence how the access is executed and what value it returns to the user; these are the ones whose consistency will be enforced before handling the access. We adopt a convention of denoting a set of conits with a script $\mathcal{F}$. The middleware is responsible for ensuring that each of $\mathsf{NE}(P, F)$, $\mathsf{OE}(P, F)$, and $\mathsf{RTE}(P, F)$ is enforced for each conit $F \in \mathcal{F}$ that an access depends on.

The three measures of consistency are enforced with different mechanisms. Both order error and real-time error are bounded purely locally—that is, P can enforce these error bounds without proactive cooperation from other processes. Numerical error is bounded by a proactive protocol that requires every process to know P's error bounds and actively cooperate to ensure they are enforced. Roughly speaking, when a data access is submitted to P by a client, P executes the following steps:

1. **Enforce OE and RTE:** For each depended-on conit F, the values $\mathsf{OE}(P, F)$ and $\mathsf{RTE}(P, F)$ are enforced using compulsory pull-based anti-entropy sessions to pull updates from remote replicas, if necessary.

2. **Apply the access locally:** The access is applied to the local replica, which in general will yield some return value.

3. **Add writes to the log:** If the access is a write access, it is tagged with the pair $(P, \mathsf{clock}(P, t))$ and pushed onto Log^P so that it may participate in write propagation.

4. **Enforce NE for remotes:** If the access is a write w, for each conit F affected by w, the system decides for each remote Q whether it is required to initiate compulsory push-based anti-entropy with Q to cooperatively bound $\mathsf{NE}(Q, F)$.

5. **Return to the user:** The request returns control back to the user, yielding the return value of the access if there was one.

Because $\mathsf{OE}(P, F)$ and $\mathsf{RTE}(P, F)$ are enforced locally, these values can be changed on a per-access basis, with each access specifying its preferred consistency levels. However, changing $\mathsf{NE}(P, F)$ requires notifying every other process about the update so they can proactively enforce it, so may be costly to change this value frequently.

5.5 Numerical Consistency

We assume each conit F is associated with a valuation function,

$$\mathcal{V}^F : \mathsf{Database\ Image} \to \mathbb{R}$$

that maps a database state to some real number representing the value of that conit. Yu and Vahdat actually *define* a conit as what we have called a valuation function. We adopt a slightly more abstract terminology and say that a conit is just an identifier such that each write is associated with a well-defined numerical weight for each conit. Intuitively, the value $\mathsf{NumWeight}(w, F)$ codifies the effect on F of applying w to the database in the sense that it should be defined by the following equation:

$$\mathsf{NumWeight}\,(w, F) \equiv \mathcal{V}^F(D + w) - \mathcal{V}^F(D). \tag{14}$$

To make sense of this equation, we must assume the value on the right-hand side is independent of the current state D of the database. This is sensible because we think of writes as containing logic to modify the database, such as incrementing or decrementing a value. This contrasts with writes that directly set the value of each data item, in which case the difference would directly depend on the item's previous value. If the change applied to F by w could depend on the state of the replica, then $\mathsf{NumWeight}\,(w, F)$ might be given as an upper bound of the above expression quantified over possible states of D instead, but have not come across a use case where this would be required.

Let $V^F_{\text{init}} \equiv V^F(D_{\text{init}})$ denote the value of F in the initial database state. Let $V^F_P \equiv V^F(D^P)$ denote the current value of F at P's replica. Let $V^F_{\text{ideal}} \equiv V^F(D_{\text{ideal}})$ represent the value of F at D_{ideal}. Now F distributes over $+$ in the sense that the following holds:

$$\mathcal{V}^F(D + w) = \mathcal{V}^F(D) + \mathsf{NumWeight}(w, F). \tag{15}$$

Combining this with Equations (7) and (11) gives

$$V^F_P = V^F_{\text{init}} + \sum\{\mathsf{NumWeight}(w, F) \mid w \in H^P\} \tag{16}$$

$$V^F_{\text{ideal}} = V^F_{\text{init}} + \sum\{\mathsf{NumWeight}(w, F) \mid w \in H_{\text{global}}\} \tag{17}$$

The correctness condition required of numerical error is that the difference between these values is bounded at all times according to the following inequality:

$$|V^F_{\text{ideal}} - V^F_P| < \mathsf{NE}\,(P, F) \tag{18}$$

Enforcing this condition requires bounding the combined weight of set of writes that P has not seen, specifically the ones that affect conit F:

$$\{w \in H_{\text{global}} \mid \mathsf{NumWeight}(w, F) \neq 0\} \setminus \mathsf{Log}^P$$

This is accomplished by requiring all *other* process to maintain approximate knowledge of the contents Log^P. Each process Q uses its estimate of Log^P to bound its own "contribution" towards the writes in H_{global} that are not in Log^P, namely the writes originating at Q that P has not been notified of yet.

5.5.1 Assumptions

Bounding numerical error requires a cooperation between all processes. For this, we make an assumption that each process knows every other process's numerical error bounds for each conit.

> **Global knowledge of error bounds**: Every process knows $\mathsf{NE}\,(P, F)$ for each conit F and each process P.

If P wishes to dynamically update its numerical error bounds for a conit F, it must invoke some mechanism to inform all other processes of this change.

In order to bound P's numerical error, each remote process Q needs to know which writes P has not seen. Specifically, Q must keep track of at least its *own* writes P has not seen. Recall that $\mathsf{Log}^P[Q]$ contains the writes P has received that originated at Q. Therefore, the set of writes originating at Q unseen by P is defined by the following set difference:

$$\mathsf{Unseen}^Q(P) \equiv \mathsf{Log}^Q[Q] \setminus \mathsf{Log}^P[Q] \tag{19}$$

Q cannot compute this set exactly because $\mathsf{Log}^P[Q]$ is a data structure maintained by P, not Q. For this reason, we require Q to maintain a conservative estimate, called its *view*, of this value. Q's view of P is denoted $\mathsf{View}^Q(P)$.

> **Approximate knowledge of remote knowledge**: Each process Q can compute a set $\mathsf{View}^Q(P)$ subject to the following invariant:
>
> $$\mathsf{View}^Q(P) \subseteq \mathsf{Log}^P[Q] \tag{20}$$
>
> Q must also implement a mechanism to advance its view by pushing updates to P, at which point $\mathsf{View}^Q(P) = \mathsf{Log}^P[Q]$. Section 5.5.2 describes two ways to implement views.

The subset relation above states that Q maintains a lower bound estimate of this set. It is a lower bound because P might have learned about writes originating at Q through anti-entropy with some intermediary process R, and Q may not know about this. The lower bound corresponds to an upper bound estimate of the updates originating at Q that P has *not* received yet, defined by the following equation:

$$\mathsf{EstUnseen}^Q(P) \equiv \mathsf{Log}^Q[Q] \setminus \mathsf{View}^Q(P) \tag{21}$$

Finally, condition (20) implies the following subset relation:

$$\mathsf{Unseen}^Q(P) \subseteq \mathsf{EstUnseen}^Q(P) \tag{22}$$

To bound numerical error, Q never allows the combined weight of writes in $\mathsf{EstUnseen}^Q(P)$ to exceed a certain threshold on a per-conit basis. As long as every process collaborates in this manner, the weight of writes unseen by P will remain within P's numerical error bounds. Section 5.5.3 describes a simple mechanism for achieving this.

5.5.2 Implementations of Views

We describe two ways to implement views for two implementations of TSAE described in Section 4.

Acknowledgment Vectors If TSAE is implemented using acknowledgment vectors, Q maintains an additional vector last^Q where $\mathsf{last}^Q[P]$ stores the value of $\mathsf{clock}\,(Q,t)$ at the time of the last push to P. Q conservatively estimates that P has not received any messages from Q (via a third party) since $\mathsf{last}^Q[P]$:

$$\mathsf{View}^Q\,(P) \equiv \{m \in \mathsf{Log}^Q\,[Q] \mid \mathsf{time}\,(m) \leq \mathsf{last}^Q[P]\} \tag{23}$$

While handling a write with timestamp $\mathsf{clock}\,(Q,t)$, after pushing messages to P, $\mathsf{last}^Q[P]$ is advanced to $\mathsf{clock}\,(Q,t)$.

Note that $\mathsf{ack}^P\,[Q]$ itself provides a conservative view of Log^Q, but it is too coarse: there is no guarantee it will advance after pushing updates to Q, since it is a lower bound of the minimum value of $\mathsf{summary}^Q\,[X]$ for *any* X.

Matrices If TSAE is implemented with summary matrices, Q estimates that P has not seen any messages from Q with timestamp newer than $\mathsf{matrix}^Q[P][Q]$:

$$\mathsf{View}^Q\,(P) \equiv \{m \in \mathsf{Log}^Q\,[Q] \mid \mathsf{time}\,(m) \leq \mathsf{matrix}^Q[P][Q]\} \tag{24}$$

The advantage of using matrix over last is that it allows Q to learn from other processes which messages originating at Q P has received, since all processes will track this information. Entries in last^Q can only be updated by Q itself, making it a coarser estimate of Q's write log, potentially causing P to push updates more often.

5.5.3 Split Weight Absolute Error

We describe the *split-weight AE* (absolute error) algorithm, which Yu and Vahdat explicate most at length in a 2000 paper [44]. For reasons explained below, we consider writes with positive and negative weights separately.

$$\mathsf{EstUnseen}|_F^{Q+}(P) \equiv \{w \in \mathsf{EstUnseen}^Q(P) \mid \mathsf{NumWeight}\,(w,F) > 0\} \tag{25}$$

$$\mathsf{EstUnseen}|_F^{Q-}(P) \equiv \{w \in \mathsf{EstUnseen}^Q(P) \mid \mathsf{NumWeight}\,(w,F) < 0\} \tag{26}$$

For each F, define the following values totaling the positive and negative weights to conit F of writes originating at Q thought to be unseen by P.

$$\mathsf{twp}|_F^Q\,(P) \equiv \sum\{\mathsf{NumWeight}\,(w,F) \mid w \in \mathsf{EstUnseen}|_F^{Q+}(P)\} \tag{27}$$

$$\mathsf{twn}|_F^Q\,(P) \equiv \sum\{\mathsf{NumWeight}\,(w,F) \mid w \in \mathsf{EstUnseen}|_F^{Q-}(P)\} \tag{28}$$

To bound $\mathsf{NE}\,(P,F)$, the intuition is that Q's estimate of numerical weight on F unseen by P must not exceed an allotted "share" of $\mathsf{NE}\,(P,F)$. There are $|\mathcal{P}|$ total processes, of which $|\mathcal{P}|-1$ can originate writes without P immediately seeing them. Therefore, the total weight of unseen messages accepted by these processes must not exceed $\mathsf{NE}\,(P,F)\,/\,(|\mathcal{P}|-1)$. When P submits a new write, P first checks whether the following conditions hold:

$$\begin{aligned}
\mathsf{twp}|_F^Q\,(P) + \mathsf{NumWeight}(w,F) &< \frac{\mathsf{NE}\,(P,F)}{|\mathcal{P}|-1} && \text{if } \mathsf{NumWeight}(w,F) > 0 \\
\mathsf{twn}|_F^Q\,(P) + \mathsf{NumWeight}(w,F) &> -\frac{\mathsf{NE}\,(P,F)}{|\mathcal{P}|-1} && \text{if } \mathsf{NumWeight}(w,F) < 0
\end{aligned}$$

If either condition is violated, then Q updates its view of P, requesting the latest value of P's summary vector and sending any unseen writes (including those not necessarily originating at Q) following a one-sided version of the usual TSAE protocol. Before the write access at Q returns to the user, Log^P will reflect all updates originating at Q before and including the one being handled, and $\mathsf{twp}|_F^Q(P)$ and $\mathsf{twn}|_F^Q(P)$ will be zero.

Lemma 5.9 (Numerical weight correctness)**.** *When following the above protocol, then each value of each conit at process will satisfy*

$$|V_{ideal}^F - V_P^F| < \mathsf{NE}(P, F).$$

Proof. The difference between the ideal value of conit F and the actual value of F at P is the sum of all weights applied to F by writes that are not in P's write log:

$$V_{\text{ideal}}^F - V_P^F = \sum_{Q \in \mathcal{P} \setminus \{P\}} \{\mathsf{NumWeight}(w, F) \mid w \in \mathsf{Unseen}^Q(P)\}$$

Because actual unseen writes are a subset of estimated unseen writes, we have the following equation:

$$\begin{aligned} &\sum \{\mathsf{NumWeight}(w, F) \mid w \in \mathsf{Unseen}^Q(P)\} \\ <&\sum \{\mathsf{NumWeight}(w, F) \mid w \in \mathsf{EstUnseen}|_F^{Q+}(P)\} \\ \equiv\ &\mathsf{twp}|_F^Q(P) \end{aligned}$$

Likewise we have the following:

$$\begin{aligned} &\mathsf{twn}|_F^Q(P) \\ \equiv&\sum \{\mathsf{NumWeight}(w, F) \mid w \in \mathsf{EstUnseen}|_F^{Q-}(P)\} \\ <&\sum \{\mathsf{NumWeight}(w, F) \mid w \in \mathsf{Unseen}^Q(P)\} \end{aligned}$$

Now as long as the invariants

$$-\frac{\mathsf{NE}(P, F)}{|\mathcal{P}| - 1} < \mathsf{twn}|_F^Q(P) \quad \text{and} \quad \mathsf{twp}|_F^Q(P) < \frac{\mathsf{NE}(Q, F)}{|\mathcal{P}| - 1}$$

are maintained for each choice of Q, then the following inequality holds:

$$-\mathsf{NE}(P, F) < V_{\text{ideal}}^F - V_P^F < \mathsf{NE}(P, F)$$

$\square$

The motivation for tracking positive and negative weights separately stems from the fact that P's view is conservative, and letting writes with positive weight "cancel out" writes with negative weight can lead to incorrect behavior. For example, suppose Q has originated writes with positive and negative weights that exactly cancel out, and suppose P has actually seen, unbeknownst to Q, all of the writes with negative weight. If Q totaled all estimated unseen weights together, it would wrongly estimate that the weight of writes unseen by P is 0, when the actual value would be some positive value that may exceed $\mathsf{NE}(P, F)$.

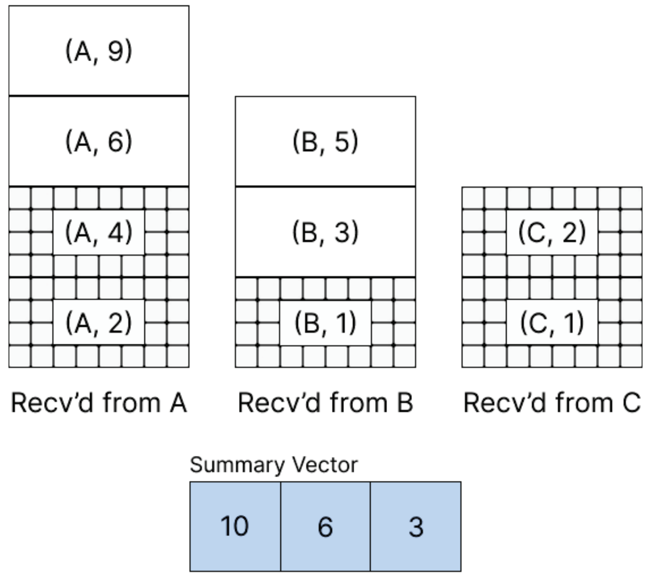


Figure 29: Write log Log^A for Example 5.10, where messages with a grid background represent A's view of Log^C.

Example 5.10. Figure 29 depicts the role of numerical weight for a process A. Messages with a grid background represent A's view of Log^C. We are assuming here that summary matrices are used to implement TSAE, so A also has a view of $\mathsf{Log}^C[B]$, though this does not play a role. A only works to bound the weight of its own writes unseen by C.

A estimates there are two messages (with timestamps $(A, 6)$ and $(A, 9)$) in $\mathsf{Log}^A[A]$ that are not in $\mathsf{Log}^C[A]$. We assume both of these updates apply positive weight to a conit F. While handling access $(A, 9)$, before returning to the user, the combined weight of the updates must be less than $\mathsf{NE}(C, F)/2$. If this limit is exceeded, A would become unavailable and engage in compulsory anti-entropy with C. Note that if A is multithreaded and can handle multiple accesses at once, then A only has to be unavailable for further accesses involving the same underlying conit.

5.5.4 Distributed Locking

A subtle but important implementation detail we have not described concerns the finer details of write propagation. Yu and Vahdat describe two possible implementations of what happens when, following the protocol in Section 5.5.3, Q decides to push updates to P. The difference between the implementations has to do with whether the protocol ensures that no concurrent writes are possible while Q handles the client's request.

Single Round Push With this implementation, Q applies the write update to its own replica immediately, then engages in anti-entropy with P (and any other required processes) to push the new update, awaits acknowledgment from all destinations, then returns to the user.

Two Round Push With this implementation, Q first acquires locks (i.e. exclusive rights to modify) covering any of the data items affected by the write on P and any other required processes. After acquiring all locks, the write is executed against D^Q and pushed to all destinations. Locks are released and the request returns to the user after receiving all acknowledgments.

The two-round protocol, a simple form of distributed two-phase locking, is required for correctness if numerical error bounds are enforced for the observed state seen write accesses w submitted to Q. Acquiring locks prevents any of the destinations from initiating their own updates while w is executed against D^Q. Otherwise, destinations may update the data while Q is handling w, and those updates may be found to precede w in the global history, making them unseen updates for w.

The single-phase protocol is appropriate when observed numerical error bounds do not have to be enforced for write updates. According to the authors, applications where writes are interchangeable may benefit from the one-round protocol.

5.5.5 Variations

Yu and Vahdat also describe two other schemes for bounding numerical error. *Compound-weight absolute error* is similar to split-weight AE, but allows positive and negative weights to cancel out. This method represents a potential tradeoff between storage space and communication, but they did not find the savings particularly compelling in their performance analysis. They also consider a scheme, *inductive relative error*, which bounds the relative error $|1 - V_P^F / V_{\text{ideal}}^F|$. It is somewhat notable that this algorithm relies only on each process' local knowledge, because it is not obvious how to efficiently bound relative error without knowing V_{ideal}^F.

5.6 Order Consistency

The write log Log^P is a linearly ordered structure, and the local replica state $D^P = D_{\text{init}} + \mathsf{Log}^P$ reflects this order. Generally, writes are added to Log^P and applied to D^P in the order they are received, instead of waiting for the totally-ordered delivery component of TSAE+TO to determine their correct order. Consequently, writes might be applied in the wrong order. This section provides a way to enforce an upper bound on the "weight" of out-of-order writes at any moment.

Periodically, following the protocol described in this section, the total-order component learns more information about the final (global) order of writes; at this time, it may discover that some of the recent updates have been applied in the wrong order. Such writes then have to be un-applied from D^P, as it were, and then reapplied in their correct order. How this rollback-and-reapply mechanism is implemented is determined by the application and its data model. The mechanism described in this section only determines when this process is necessary to avoid violating the application's consistency requirements.

Writes accepted in H^P whose final order has been confirmed are said to be committed. Other writes are said to be tentative. This distinction divides the local history into two contiguous segments:

$$H^P = \mathsf{Committed}^P + \mathsf{Tentative}^P \tag{29}$$

The out-of-order error $\mathsf{OE}(P, F)$ is enforced as an upper bound on the weight of the tentative segment, relative to the conit F. Before defining this value formally, we describe a running example, adapted from an application described by Yu and Vahdat [42], which we will use to demonstrate out-of-order error and write commitment.

Example 5.11. Suppose P and Q are two web servers independently processing client requests to reserve seats on a flight. To improve performance, they do not maintain strong consistency. Now suppose both process separate reservations for seat $A1$ at the same time, denoted w_P and w_Q. This results in inconsistent states, $D_{\text{init}} + w_P$ and $D_{\text{init}} + w_Q$, respectively.

Eventually, P and Q learn about each other's updates through anti-entropy and commit them in an agreed order. Assume P's update is ordered first. In this case, P transitions to $D_{\text{init}} + w_P + w_Q$, while Q rolls back its state and reapplies its write as follows:

$$D_{\text{init}} + w_Q \xmapsto{\text{rollback}} D_{\text{init}} \xmapsto{\text{reapply}} D_{\text{init}} + w_P + w_Q$$

Recall that w_P and w_Q represent transactions that may include business logic. The final state $D_{\text{init}} + w_P + w_Q$ might be interpreted as, "Seat $A1$ is reserved for P's client, while Q's client experiences a double-booking error." The rollback process may trigger notifications to Q's client about the cancellation, along with issuing a refund (and possibly a compensation fee).

The order weight $\mathsf{OrderWeight}\,(w, F)$ of a write w defines the relative cost of rolling back a tentative update, which can be thought of as a liability. In Example 5.11, accepting w_Q in a tentative state represented a liability for the airline. One way to bound this liability is by defining a single conit, say F_{flight}, to represent the seats on the flight. Then, a natural policy would be to define each reservation on the flight to apply $\mathsf{NumWeight}\,(w, F_{\text{flight}}) = 1$ (or whatever number of seats the reservation is for) and set $\mathsf{OrderWeight}\,(w, F_{\text{flight}})$ to the airline's cost to cancel the reservation and refund the reservation holder. We will now walk through how such an application would execute depending on how the bounds are set.

Suppose the airline wishes to avoid double-booking. Setting $\mathsf{OE}(Q, F_{\text{flight}}) = 0$ would suffice to ensure that Q never accepts a reservation without knowing its final order in the global history, meaning Q never incorrectly tells a client that their reservation has been successful. On the other hand, suppose the airline is willing to risk double-booking seats to support application performance (perhaps P and Q are located on opposite sides of the world). In this case, $\mathsf{OE}(Q, F_{\text{flight}})$ would be set to the maximum potential cost of issuing refunds the airline is willing to tolerate from server Q. Effectively, Q would be free to issue some reservations without immediately checking for seat conflicts, so long as an allowable dollar amount of potential refunds is not exceeded.

An indirect way to bound this sort of liability would be for Q to enforce $\mathsf{NE}(Q, F_{\text{flight}}) = 0$. Indeed, if the two-phase locking policy is used (see Section 5.5.4), Q would not accept a reservation without being aware of a conflicting reservation at P. However, this would be enforced with a push-based protocol where P immediately notifies Q of every new reservation, which may be inefficient. Furthermore, this metric does not provide a natural way of allowing Q to issue a controlled number of tentative reservations. The protocol to bound order error, described below, is pull-based and would only require communication with P when the cost of Q's tentative reservations exceeds its allowable bounds. One or the other strategy may be substantially more efficient depending on the application characteristics.

5.6.1 Out-of-Order Error

We now define out-of-order error rigorously.

Definition 5.12. Given two histories H and H', let their *intersection* or *greatest common prefix*, $H \cap H'$, be the set of all accesses they agree on since their beginning

Definition 5.13. The *errant history* of an access a submitted to P is defined as all of the writes at the tail end of $\mathsf{PH}_{\text{obs}}(a)$, beginning at the first write where $\mathsf{PH}_{\text{obs}}(a)$ does not agree with $\mathsf{PH}_{\text{ideal}}(a)$:

$$\mathsf{PH}_{\text{obs}}(a) - (\mathsf{PH}_{\text{obs}}(a) \cap \mathsf{PH}_{\text{ideal}}(a))$$

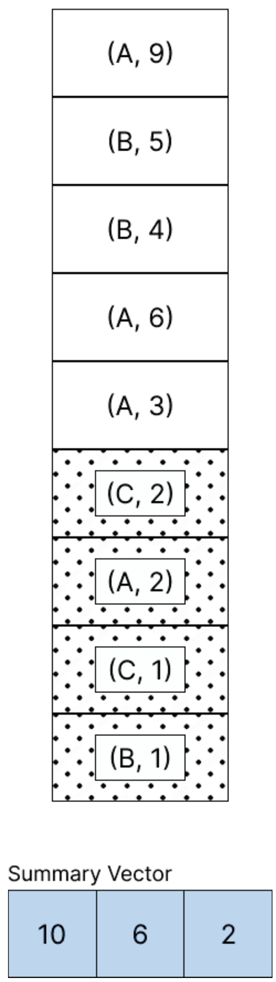
As an exercise, the reader may which to check that the following relation holds, where $H \subseteq H'$ means H is a prefix of H':

$$\mathsf{Committed}^P \subseteq \mathsf{PH}_{\text{obs}}(a) \cap \mathsf{PH}_{\text{ideal}}(a) \tag{30}$$

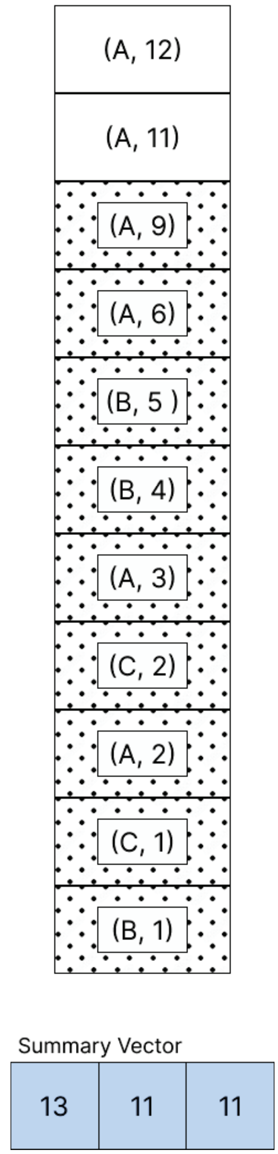
This follows because the committed segment of H^P is always a prefix of both the ideal history H_{global} and the observed history $\mathsf{PH}_{\text{obs}}(a)$. Equation (30) implies that the errant history is a suffix of the tentative writes. These facts are demonstrated in the next example.

Example 5.14. Figure 30 depicts two write logs where committed writes are shown with a dotted grid pattern. In Figure 30a, five writes are uncommitted. No write with timestamp greater than or equal to 3 can be committed, since the minimum timestamp in the summary vector is only 2. The final order of these accesses is shown in Figure 30b. For the next access submitted to A, with tag $(A, 11)$, the errant history consists of all accesses after and including $(A, 6)$, the point at which the write log deviates from the ideal history.

$$\begin{aligned}
\mathsf{PH}_{\text{obs}} \cap \mathsf{PH}_{\text{ideal}} &= \{(B,1)\ldots(A,3)\} \\
\mathsf{PH}_{\text{obs}} - (\mathsf{PH}_{\text{obs}} \cap \mathsf{PH}_{\text{ideal}}) &= \{(A,6),(B,4),(B,5),(A,9)\}
\end{aligned}$$



(a) Log^A shown with 5 uncommitted writes



(b) Log^A shown at a later time

Figure 30: Logs for Example 5.14 where committed writes are drawn with a dotted background

Let r be a read access submitted to P with a dependency on conit set $\mathcal{F}$. Let $\mathsf{Tentative}^P$ represent the uncommitted messages in Log^P. As a set, the errant history is a subset (actually a suffix) of the tentative writes.

$$\mathsf{PH}_{\text{obs}}(a) - (\mathsf{PH}_{\text{obs}}(a) \cap \mathsf{PH}_{\text{ideal}}(a)) \subseteq \mathsf{Tentative}^P \tag{31}$$

Therefore, the tentative writes can be used as an upper bound estimate of which writes may be out of order.

For a read access that depends on conits in a set $\mathcal{F}$, the relative order of updates applied to D^P only matters if they affect (i.e. apply non-zero order weight to) one or more of the conits in $\mathcal{F}$. (This can be considered an axiomatic constraint on how applications should define conits and order weight.) This means the errant history actually overstates the difference between PH_{obs} and $\mathsf{PH}_{\text{ideal}}$. For instance, if they only differ in the relative order of updates applied to conits that do not affect $\mathcal{F}$, then a read depending on $\mathcal{F}$ should be considered to observe 0 order error.

Definition 5.15 (Write-order projection)**.** Let $H|_{\mathcal{F}}$ denote the restriction of H to $\mathcal{F}$, consisting of just the writes w such that $\mathsf{OrderWeight}(w, F) \neq 0$ for at least one F in $\mathcal{F}$.

Definition 5.16. For an access a depending on a set of conits $\mathcal{F}$, the set of observed *out-of-order* accesses to $\mathcal{F}$, $\mathsf{OOO}\,(a, \mathcal{F})$ is the set of all writes affecting any conit in $\mathcal{F}$ that have been applied in an order deviating from their final order relative to each other:

$$\mathsf{OOO}\,(a, \mathcal{F}) = \mathsf{PH}_{\text{obs}}(a)|_{\mathcal{F}} - (\mathsf{PH}_{\text{obs}}(a)|_{\mathcal{F}} \cap \mathsf{PH}_{\text{ideal}}(a)|_{\mathcal{F}}) \tag{32}$$

The reader may want to verify with an example that computing the errant history and then restricting to writes that affect $\mathcal{F}$, yielding the sequence

$$(\mathsf{PH}_{\text{obs}}(a) - (\mathsf{PH}_{\text{obs}}(a) \cap \mathsf{PH}_{\text{ideal}}(a)))\,|_{\mathcal{F}}$$

overcounts the number of out-of-order updates if only the relative order of writes affecting $\mathcal{F}$ is considered important.

The order error for conit F of an access a depending on conits in $\mathcal{F}$ is the total order weight of out of order writes.

Definition 5.17 (Order error)**.** Let a be an access depending on a conit set $\mathcal{F}$. The *order error* relative to the conit F is the sum of $\mathsf{OrderWeight}\,(a, F)$ for all writes w that are in the out-of-order set for $\mathcal{F}$.

$$\mathsf{OrderError}\,(a, F) \equiv \sum \{\mathsf{OrderWeight}\,(w, F) \mid w \in \mathsf{OOO}\,(a, \mathcal{F})\} \tag{33}$$

The out-of-order set is always a subset of the uncommitted writes. So, P can estimate the order error of an access by totaling the order weight of accesses to $\mathcal{F}$ that have not been committed, yielding the following value:

$$\mathsf{EstOrderError}\,(a, F) \equiv \sum \{\mathsf{OrderWeight}\,(w, F) \mid w \in \mathsf{Tentative}^P|_{\mathcal{F}}\} \tag{34}$$

Now $\mathsf{OOO}\,(a, \mathcal{F}) \subseteq \mathsf{Tentative}^P|_{\mathcal{F}}$ implies $\mathsf{OrderError}\,(a, F) \leq \mathsf{EstOrderError}\,(a, F)$. Therefore it suffices for P to bound the estimated order error of each access.

5.6.2 Implementation

To bound out-of-order error, we assume that processes, when required, can invoke a write commitment mechanism to learn the final order of their tentatively accepted updates.

> **Ability to commit writes**: While handling an access, each process has a mechanism to determine the final order of any uncommitted writes in its write log.

For the TSAE+TO implementation, commitment is implemented with a pull-based approach. Recall that P can commit writes by pulling messages through anti-entropy to advance its commit line ($\min_{X \in \mathcal{P}} \left(\mathsf{summary}^P [X]\right)$) far enough for the total ordering component to decide the final value of all writes in its log, meaning the commit line must be greater than the maximum timestamp of a write in the log. When P pulls enough messages from remotes to decide the final order of its tentative updates, any errant updates can be rolled back and reapplied in their final order.

Let t be the greatest timestamp of any message in P's write log. To guarantee that P can advance its commit line enough to commit the write, it is enough to ensure that $\mathsf{summary}^P [Q]$ is greater than t after engaging in anti-entropy with each Q. This is true of physically synchronized clocks, since $\mathsf{summary}^P [Q]$ will reflect the time at which P initiated contact with Q. This is also true if processes maintain simple scalar logical clocks, since $\mathsf{summary}^P [Q]$ would be set to the value of Q's scalar clock after being contacted by P, which will be greater than P's clock, itself an upper bound of the timestamp of any message in the log.

Let P accept an access a from a client with a dependency on the conits in $\mathcal{F}$. The correctness condition of this metric is that the following inequality holds:

$$\mathsf{OrderError}\,(a, F) < \mathsf{OE}(P, F) \tag{35}$$

While handling a, for each conit F that a depends on, if $\mathsf{EstOrderError}(a, F)$ exceeds $\mathsf{OE}(P, F)$, P engages the write commitment procedure to commit enough of its tentative updates to make the inequality hold. This process is repeated for each conit F that a depends on.

Note that because P does not require proactive cooperation from other processes in $\mathcal{P}$, the value of $\mathsf{OE}(P, F)$ can be set uniquely for each access submitted to P. This allows controlling the risk of an out-of-order access on a per-access basis. For example, a preferred client of P may enforce 0 order error for any conits it depends on, ensuring that client never risks an canceled reservation.

Example 5.18. Suppose in Figure 30a that an access submitted to A at time $t = 11$ is timestamped $(A, 11)$. Suppose all writes apply unit order weight to a conit F. If $(A, 11)$ depends on F with an allowable order error of less than 5, the number of uncommitted writes, then the update will be executed until A pulls writes from B and C so that all messages up to $(A, 9)$ can be committed. These anti-entropy sessions increase $\mathsf{summary}^A [B]$ and $\mathsf{summary}^A [C]$ to 11. Figure 30b shows a possible state at a later time $t = 13$, after accepting another write with timestamp $(A, 12)$

5.7 Real Time Staleness

The *real time staleness* metric allows each process P to bound the maximum amount of time between an access affecting a conit being issued and P seeing that access. Here we rely on the assumption that $\mathsf{summary}^P$ stores physical timestamps from loosely synchronized physical clocks, though this assumption can be weakened.

Definition 5.19. The *real-time staleness* of an access a relative to conit F is the value

$$\mathsf{start}\,(a) - \min \left\{ \mathsf{stop}\,(w) \quad \middle| \quad \begin{array}{c} w \in \mathsf{PH}_{\text{ideal}} - \mathsf{PH}_{\text{obs}} \\ \text{and } \mathsf{NumWeight}\,(w, F) \neq 0 \\ \text{and } w \text{ externally precedes } a \end{array} \right\}$$

The correctness condition for real-time staleness is that the above value is less than $\mathsf{RTE}\,(P, F)$ for any access originating at P with a read dependency on F.

5.7.1 Implementation

While handling an access with a read dependency on F, submitted at real time t, P checks for each $X \in \mathcal{P}$ whether $|\mathsf{clock}\,(P, t) - \mathsf{summary}^P\,[X]\,| < \mathsf{RTE}\,(P, F)$ is true for each X. If it is not true, then P engages in a pull-based anti-entropy session with X, at which point $\mathsf{summary}^P\,[X]$ has value $\mathsf{clock}\,(X, t')$ for some $t' \approx t$. Note that the assumption of loose synchronization implies $\mathsf{clock}\,(X, t') \approx \mathsf{clock}\,(P, t)$.

This protocol ensures that the original access, timestamped with value $\mathsf{clock}\,(P, t)$, will observe the effect of all writes affecting F with timestamps less than $\mathsf{clock}\,(P, t) - \mathsf{RTE}(P, F)$. The assumption of external precedence owes to the corner case scenario where processing the access takes a long time, during which time some other process, whose writes were already pulled by P, originates a new write that finishes before P's write by more than $\mathsf{RTE}\,(P, F)$. Such a write has no external precedence over P's write, although it may be ordered before it in the global order.

A pull-based protocol may seem wasteful because P may poll remotes for updates even if when they do not have any new writes. However, an approach where remotes push updates to P cannot bound real-time staleness without an upper bound on the time it takes to push messages across the network to P. The pull-based approach does not suffer the same problem—high network latencies will cause P to block for a long time, but will preserve the intended correctness property.

If clocks are not loosely synchronized, an alternative implementation strategy is for P to maintain a vector $\mathsf{last}^P_{\mathsf{RT}}$ where $\mathsf{last}^P_{\mathsf{RT}}[X]$ stores the value $\mathsf{clock}\,(P, t)$ of the last time P was on the receiving end of an anti-entropy session directly with X. Then P enforces consistency by using $\mathsf{last}^P_{\mathsf{RT}}$ in place of $\mathsf{summary}^P\,[X]$ above. This approach only compares values from P's clock; synchronization is not required assuming $\mathsf{clock}\,(P)$ runs at a constant rate. However, this approach has the downside that, unlike $\mathsf{summary}^P\,[Q]$, the value of $\mathsf{last}^P_{\mathsf{RT}}[Q]$ is not updated based on information indirectly learned during anti-entropy with a third party.

5.8 Correctness Properties

Section 5.5.4 described two implementations of write propagation that differ in their guarantees. The stronger but less efficient method involves a two-phase locking procedure where, during write propagation, remote locks covering all of the affected data items are first acquired from all hosts where one intends to send write updates. Once all locks have been acquired, all necessary writes are sent. While locks are held, reads from affected conits are blocked on those remotes. When the writes are propagated, locks are released. This two-phase locking mechanism, which is also a conventional method to implement linearizability, ensures that the propagated write operations appear to take effect at all destinations at the same time. When writes are propagated this way, the numerical and order weights of all accesses are properly bounded with respect to their ideal values in a strongly consistent history.

Theorem 5.20 (Yu and Vahdat)**.** *Assuming the two-phase locking policy is used during compulsory write propagation, there is an ECG history such that the observed numerical, order, and real-time error of every access is correctly bounded with respect to its ideal return value in the ECG history.*

The following result is Theorem 4.1 from [42]. Note that "strict serializability" here is essentially synonymous linearizability in the context of replicated databases. (The condition also implies serializable isolation of transactions, which is a concept specific to the database context.)

Theorem 5.21. *If a conit is defined for each data item, and an access depends on each conit it reads or writes, if the allowable numerical error and order error on all conits is* 0*, and all writes have unit numerical and order weight on all affected conits, then the system provides strict serializability.*

If the one-round protocol is used where remote locks are not acquired before pushing writes, then some replicas appear to see new writes before others. That is, this policy does not enforce atomic visibility —updates at different replicas, appear to occur at different physical times instead of all at once. This is a weaker consistency model but may be valuable for certain kinds of information, particularly for applications where accepting new updates is always valuable, even if this update has not propagated to others yet. Otherwise, the two-round protocol would require all replicas to be notified before the new information can affect future read requests.

5.9 Extensions

One can imagine various ways that the conit model can be augmented with additional capabilities. These topics are outside the scope of this memo but offered for future consideration.

Dynamic bounds We imagine dynamically changing error bounds automatically in response to operational factors. For example, if the network is badly congested, the application may temporarily relax its error bounds. Returning to the VLAT example in Section 2.1, heavy smoke may cause an application to relax the real-time staleness bounds for information about firefighters' locations on the ground. Correspondingly, the application may enforce a wider margin for authorizing VLAT drops, requiring the location estimates to be an even greater distance outside the drop zone to compensate for the increased uncertainty. This can resolve the discontinuity highlighted in that section, where poor network performance might require grounding VLATs to enforce an overly rigid safety policy that cannot be adjusted for ground conditions.

Because real-time staleness and order error are bounded by pull-based anti-entropy, the user can dynamically change the error bounds for each access. However, numerical error is bounded by a push-based approach that requires every process to track all other processes' error bounds. One might follow Golding's approach [15] to extend the model with dynamic group membership and provide a consensus mechanism for adjusting numerical error bounds.

Dynamic conits The framework we have described above assumes the set of conits is fixed in advance. Besides tuning bounds dynamically, we can imagine situations where new conits need to be created on-the-fly. For instance, data related to a new wildfire that has emerged may require forming new conits to set consistency bounds.

Donkervliet's master's thesis [12] explored the subject of dynamic conit creation in the context of massive multiplayer online games, particularly Minecraft. In that work, new conits may be associated with newly encountered objects in an area, and their consistency bounds tuned as the user approaches them, exploiting a form of locality to allocate network resources for information where inconsistency would most readily be perceived by the player. Adapting their *dyconits* (dynamic conits) framework to the wide-area tactical environment may be worthwhile. It is notable that games like Minecraft, where players navigate three-dimensional environments and communicate over a network with sometimes perceptible delays, share some similarities with real-world environments, such as the fact nearby users are likely to share an interest in the same conits.

Interaction with the Network We mentioned in Section 2.3.2 our expectation of a tighter, more complex interaction between the network and application layers in this environment because of the need to optimize scarce network resources. Because conits allow quantifying divergence, and therefore the relative importance of an update, one way this might be realized is by making the network conit-aware. Network packets, or DTN bundles [33], could be specially marked as containing database updates alongside any metadata such as the weight of the update to various conits. This information could then be used by the network for quality-of-service purposes.

Such usage may run contrary to a general wisdom that networking protocols should be agnostic to the content of a message—routers for example should only inspect IP packet headers but not the data payload. This sort of usage may be justified in our setting because of a heightened requirement to optimize network resources, potentially at the cost of blurring the line between the network and application layers.

Modifying a network protocol to optimize a particular middleware or application is not a lightweight task, particularly since network drivers are often embedded into an operating system kernel or into hardware. Software-defined networking [29], a trend that broadly raises network protocols to the level of ordinary software development, is likely to be highly useful for this kind of experimentation. Using a language such as P4 [4] to implement a conit-aware extension of delay-tolerant networking protocols, for instance building on work by Ta et al. [36], would offer the ability to prototype designs without needing to design special-purpose hardware or replace low-level drivers.

6 Conclusion

The increasing severity of wildfires and other natural disasters has underscored the need for more advanced and reliable communication systems for first responders. Modern tools like the Team Awareness Kit (TAK) for situational awareness at the edge, as well as more data-intensive centralized processes like NASA's Fire Information for Resource Management System (FIRMS), have demonstrated the need and the utility of large-scale complex systems for collecting, processing, and distributing data among agents operating in adverse environments. Traditional communication models, designed for more stable environments, cannot necessarily meet the demands of such dynamic and critical scenarios.

At a low level, the challenges of these environments can be addressed through innovations in hardware such as mesh radios operating with protocols optimized for mobile ad-hoc and disruption-heavy networks. However, these non-traditional network architectures and extreme operational conditions raise higher-level questions about how to design distributed applications that communicate over these networks. We explored how applications that rely on strong memory consistency models, such as linearizability or sequential consistency, are not a good fit for this sort of use case. This implication stems largely from the locality principle in Section 2.2, which suggests that the quality of communication between distributed agents—whether measured by message latency, packet loss, or some other metric—is strongly influenced by the geographical distance between them.

Strong consistency models are well-suited to scenarios where a centralized database can be accessed through a small number of dedicated servers, but locality suggests that agents at the tactical edge will face difficulty maintaining communication with central servers. Alternatively, for resilience and efficiency, multiple replicas of a shared database can be maintained and deployed at strategic locations in the network. However, the high coordination overhead required to keep these replicas consistent with respect to strong memory models is prohibitive for the networks under consideration. We concluded that weaker consistency models are called for in these settings.

Brewer's CAP theorem, a cornerstone result in distributed systems theory, seemingly poses a major challenge when strong consistency is unattainable. A simplistic interpretation of the theorem suggests that when strong consistency is sacrificed in favor of system performance, system designers are left with little control over the relative consistency between different replicas of common data. This presents an uncomfortable dilemma, because the appreciable safety concerns in disaster response environments demand control over the consistency of shared information. This is because a common operating picture is a prerequisite to effective coordination between agents. The CAP theorem seems to suggest that the application designer is stuck between a rock and a hard place.

Fortunately, the CAP theorem is not as prohibitive as it may seem. The continuous consistency framework presented in Section 5—though it guarantees neither the 'C' nor the 'A' in the CAP theorem—presents a way through this impasse. By formally quantifying the divergence between replicas with respect to units of consistency, and then providing efficient mechanisms for estimating this divergence using only local information, applications can track the relative consistency of distributed replicas with respect to different metrics and react intelligently to these values. The framework we presented is dynamic, providing ways for applications to adjust their position along the spectrum between entirely favoring consistency and entirely favoring system availability. How best to exploit these mechanisms, such as designing policies that automatically tune the application based on observed system performance, remains an exciting and largely unexplored topic that will benefit from future work along theoretical and applied fronts.

In conclusion, we have given a broad overview of the key challenges in distributed systems in the context of natural disaster environments, with a particular focus on memory consistency in contexts where the network is unreliable and message-passing is subject to significant delays. This content provides a foundation for building resilient, high-performance systems tailored to the needs of disaster response in the 21$^{\text{st}}$ century.

Bibliography

References

1. Mustaque Ahamad et al. "Causal Memory: Definitions, Implementation, and Programming". In: *Distributed Computing* 9 (1995), pp. 37–49. URL: `https://api.semanticscholar.org/CorpusID:6435056`.
2. Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/0020-0190(85)90056-0`.
3. Tim Berners-Lee, Roy T. Fielding, and Larry M Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. Jan. 2005. DOI: `10.17487/RFC3986`.
4. Pat Bosshart et al. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: `10.1145/2656877.2656890`.
5. Eric Brewer. "CAP Twelve Years Later: How the "Rules" Have Changed". In: *Computer* 45.2 (2012), pp. 23–29. DOI: `10.1109/MC.2012.37`.

6. Philip Brey and Johnny Hartz Søraker. "Philosophy of Computing and Information Technology". In: *Philosophy of Technology and Engineering Sciences*. Ed. by Anthonie Meijers. Handbook of the Philosophy of Science. Amsterdam: North-Holland, 2009, pp. 1341–1407. DOI: https://doi.org/10.1016/B978-0-444-51667-1.50051-3.

7. California Department of Forestry and Fire Protection. *Firefighter Injuries and Fatality: August 13, 2018, Mendocino Complex (Ranch Fire)*. California, USA: California Department of Forestry and Fire Protection, Aug. 13, 2018. URL: https://web.archive.org/web/20180918124244/https://wildfiretoday.com/documents/Green_Sheet_Retardant_Drop_Fatality.pdf.

8. California Department of Forestry and Fire Protection. *Monument Fire*. Available at https://www.fire.ca.gov/incidents/2021/7/30/monument-fire/. Oct. 14, 2022. (Visited on 01/08/2024).

9. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (International Computer Science)*. 4th rev. ed. Addison-Wesley Longman, Amsterdam, 2005. ISBN: 0321263545.

10. J.D. Day and H. Zimmermann. "The OSI Reference Model". In: *Proceedings of the IEEE* 71.12 (1983), pp. 1334–1340. DOI: 10.1109/PROC.1983.12775.

11. Dereck Orr. *How 9/11 Changed Me and First Responder Communications*. Available at https://www.nist.gov/blogs/taking-measure/how-911-changed-me-and-first-responder-communications. NIST, Aug. 20, 2021.

12. Jesse Donkervliet. "Design and Experimental Evaluation of a System based on Dynamic Conits for Scaling Minecraft-like Environments". Available at https://resolver.tudelft.nl/uuid:4045d6a2-87ae-4397-8898-8e992fa0652c. Master's thesis. Example City, CA: Delft University of Technology, Jan. 2018.

13. A. Fox and E.A. Brewer. "Harvest, Yield, and Scalable Tolerant Systems". In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. 1999, pp. 174–178. DOI: 10.1109/HOTOS.1999.798396.

14. Seth Gilbert and Nancy A. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

15. Richard Andrew Golding. "Weak-Consistency Group Communication and Membership". UMI Order No. GAX93-12435. PhD thesis. USA, 1992.

16. Harry Markley. *Remembering 9/11 and How It Changed Public Safety communications*. Available at https://www.firstnet.gov/newsroom/blog/remembering-911-and-how-it-changed-public-safety-communications. FirstNet, Sept. 7, 2021.

17. Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

18. Jesse Stark/“Free-Range-Human Project”. *Mechanized Wildland Firefighting. Ep.64*. https://www.youtube.com/watch?v=4F2dDKMgAME. Youtube, May 2022.

19. Alexander Kott, Ananthram Swami, and Bruce J. West. “The Internet of Battle Things”. In: *Computer* 49.12 (2016), pp. 70–75. DOI: 10.1109/MC.2016.355.

20. Roman Kovalchukov et al. “DECT-2020 New Radio: The Next Step toward 5G Massive Machine-Type Communications”. In: *IEEE Communications Magazine* 60 (June 2022), pp. 58–64. DOI: 10.1109/MCOM.001.2100375.

21. Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511805318.

22. Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

23. Stew Magnuson. “’Coin of the Realm’: Military ’Swimming In Sensors and Drowning in Data’”. In: *National Defense* 94.674 (2010), pp. 36–38. ISSN: 00921491, 19433115. URL: http://www.jstor.org/stable/45370790 (visited on 11/10/2023).

24. R. Martí et al. “Providing Early Resource Allocation During Emergencies: The Mobile Triage Tag”. In: *Journal of Network and Computer Applications* 32.6 (2009), pp. 1167–1182. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2009.05.006.

25. Friedemann Mattern. “Virtual Time and Global States of Distributed Systems”. In: *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*. Ed. by M. Cosnard et. al. Elsevier Science Publishers B. V., 1989, pp. 215–226. URL: https://api.semanticscholar.org/CorpusID:7517210.

26. D.L. Mills. *Network Time Protocol (version 2) Specification and Implementation*. RFC 1119. Sept. 1989. DOI: 10.17487/RFC1119.

27. David Mills. *Measured Performance of the Network Time Protocol in the Internet system*. RFC 1128. Oct. 1989. DOI: 10.17487/RFC1128.

28. Álvaro Monares et al. “Mobile Computing in Urban Emergency Situations: Improving the Support to Firefighters in the Field”. In: *Expert Syst. Appl.* 38.2 (Feb. 2011), pp. 1255–1267. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2010.05.018.

29. Jéferson Nobre et al. “Toward Software-Defined Battlefield Networking”. In: *IEEE Communications Magazine* 54 (Oct. 2016), pp. 152–157. DOI: 10.1109/MCOM.2016.7588285.

30. President's Council of Advisors on Science and Technology. *Report to the President: Modernizing Wildland Firefighting to Protect Our Firefighters*. Washington, D.C.: Executive Office of the President, Feb. 2, 2023. URL: https://web.archive.org/web/20230401084148/https://www.whitehouse.gov/wp-content/uploads/2023/02/PCAST_Wildfires-Report_Feb2023.pdf.
31. R. Stickney. "Cal Fire YouTube Clip Shows Power of Fire Retardant Drop". In: *NBC San Diego* (Aug. 26, 2019). (Visited on 10/07/2022).
32. Marguerite Reardon. "Post 9/11: Can We Count on Cell Networks?" In: *CBS News* (Sept. 7, 2011). URL: https://www.cbsnews.com/news/post-9-11-can-we-count-on-cell-networks/ (visited on 01/08/2024).
33. Joel J.P.C. Rodrigues and Vasco N.G.J. Soares. "An Introduction to Delay and Disruption Tolerant Networks (DTNs)". In: *Advances in Delay-Tolerant Networks (DTNs) (Second Edition)*. Ed. by Joel J.P.C. Rodrigues. Second Edition. Woodhead Publishing Series in Electronic and Optical Materials. Woodhead Publishing, 2021, pp. 1–20. ISBN: 978-0-08-102793-6. DOI: https://doi.org/10.1016/B978-0-08-102793-6.00001-1.
34. J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end arguments in system design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402.
35. Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. USA: McGraw-Hill, Inc., 1994. ISBN: 007057572X.
36. Dominick Ta, Stephanie Booth, and Rachel Dudukovich. "Towards Software-Defined Delay Tolerant Networks". In: *Network* 3.1 (2023), pp. 15–38. ISSN: 2673-8732. DOI: 10.3390/network3010002.
37. United States Department of Agriculture. *Standards for Airtanker Operations*. July 2019. URL: https://www.fs.usda.gov/sites/default/files/2019-09/standards_for_airtanker_operations_-_final_-_2019_approved_0.pdf.
38. WGNO Web Desk. "Cajun Navy says No Shots Fired at Rescuers; Statements Made by Another Rescue Group". In: *WGNO* (Aug. 29, 2017). URL: https://wgno.com/us-world-news/update-cajun-navy-says-no-shots-fired-at-rescuers-statements-made-by-another-rescue-group/ (visited on 09/17/2024).
39. Wings Over the Rockies Air and Space Museum. *Aerial Firefighting Progress — Behind the Wings on PBS*. Available at https://www.youtube.com/watch?v=CWjfpMORGSQ&t=138. Youtube, Nov. 18, 2021.
40. Haifeng Yu and Amin Vahdat. "Building Replicated Internet Services using TACT: a Toolkit for Tunable Availability and Consistency Tradeoffs". In: *Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000*. 2000, pp. 75–84. DOI: 10.1109/WECWIS.2000.853861.

41. Haifeng Yu and Amin Vahdat. "Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication". In: *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*. IEEE Computer Society, 2001, pp. 429–438. DOI: 10.1109/ICDSC.2001.918973.

42. Haifeng Yu and Amin Vahdat. "Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services". In: *ACM Trans. Comput. Syst.* 20.3 (Aug. 2002), pp. 239–282. ISSN: 0734-2071. DOI: 10.1145/566340.566342.

43. Haifeng Yu and Amin Vahdat. "Design and Evaluation of a Continuous Consistency Model for Replicated Services". In: *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000.

44. Haifeng Yu and Amin Vahdat. "Efficient Numerical Error Bounding for Replicated Network Services". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 123–133. ISBN: 1558607153.