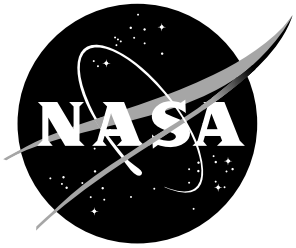
NASA/TM–2022–XXXXXX



A Survey of Distributed Systems Challenges for Wildland Firefighting and Disaster Response

Lawrence Dunn
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA
Alwyn Goodloe
NASA Langley Research Center, Hampton, Virginia

December 2022

NASA STI Program. . . in Profile

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 NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection 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:

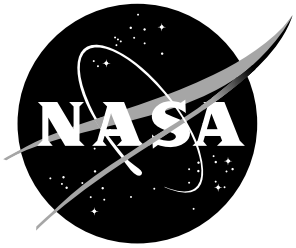
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference 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`
- E-mail your question to `help@sti.nasa.gov`
- Phone the NASA STI Information Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2022–XXXXXX



A Survey of Distributed Systems Challenges for Wildland Firefighting and Disaster Response

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

December 2022

Acknowledgments

The work was conducted during a summer internship 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.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

The System Wide Safety (SWS) program has been investigating how crewed and uncrewed aircraft can safely operate in shared airspace. Enforcing safety requirements for distributed agents requires 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. We survey topics ranging from consistency models and network architectures to data replication and data fusion, in each case focusing on the practical relevance of topics in the literature to the sorts of scenarios and challenges we expect from our use case.

Contents

1 Introduction

Civil aviation has traditionally focused primarily on the efficient and safe transportation of people and goods via the airspace. Despite the inherent risks, the application of sound engineering practices and conservative operating procedures has made flying the safest mode of transport today. Now the desire not to compromise this safety makes it difficult to integrate unmanned vehicles into the airspace, accomodate emerging applications, and keep pace with unprecedented recent growth in commercial aviation. To that end, the System Wide Safety (SWS) project of the NASA Aeronautics' Airspace Operations and Safety Program (AOSP) has been investigating technologies and methods by which crewed and uncrewed aircraft may 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 aims to be self-contained and accessible to a technical audience without a deep background in distributed systems. Our primary motivating use cases have been taken from civil emergency response scenarios, especially wildfire suppression and hurricane relief, primarily for three reasons. First, improved technology for wildfire suppression, especially related to communications and data sharing, is frequently cited as a national priority [11]. Second, the rules for operating in the US national airspace are typically relaxed during natural disasters and relief efforts, so this is a suitable setting for testing new technologies. Finally, this setting is an excellent microcosm for the sorts of general challenges faced by other, non-emergency applications.

If there is a central theme uniting the sections of this manuscript, it is *continuity* in the sense considered by topology.[1] The systems we consider will be subject to harsh operating conditions that limit how well they can perform—for example, wireless communication is typically less reliable during inclement weather. To build a system that is predictable (clearly a prerequisite for safety), one must ensure the system can perform reasonably well under a wide variety of adverse conditions. In other words, the behavior of a safe system should in some sense be a *continuous* function of its inputs and environment. This sort of robust design is particularly challenging because distributed systems designers are forced to make delicate tradeoffs between competing objectives, most famously between performance and consistency, the topic of Section 3.

1.1 Summaries of the sections

Sections 2–4 contain background material on disaster response, distributed systems, and the specifics of our use case. The critical takeaway of these sections is that system-wide safety is, at least in part, a computer science problem, indeed a software problem, and not "just" a matter of engineering better hardware. Sections 5–7 survey particular topics from the distributed systems literature, proceeding from lower-level considerations to higher-level ones; these sections may be read independently of each other. Below we summarize each section.

[1] For a typical introductory textbook see [9].

Section 2 starts with a pragmatic summary of disaster response and some of the relevant computing challenges in that setting. We aim to justify and explain the role of distributed systems theory in system-wide safety by citing real examples encountered in disaster response scenarios.

Section 3 is an introduction to distributed systems, culminating in a illustrative result: the "CAP" theorem(s) for the atomic and sequential consistency models (Theorems 3.2 and 3.3, respectively). CAP is considered a "negative" result, meaning it proves something cannot be done. The CAP theorem proves that strong consistency for a distributed system makes systemwide network performance an upper bound on the availability of a system to do useful work for clients, which for our purposes is an unacceptable restriction. The practical significance of CAP is that in emergency response environments, agents will always act with incomplete information about the global system, a key motivation for Section 6.

Section 4 refines our assumptions and identifies desirable properties of systems for our use case. We use these points to frame the discussion of systems and protocols in subsequent sections.

Section 5 examines networking considerations. Our vision of future emergency communication networks integrates concepts from delay/disruption-tolerant networks (DTN) and mobile ad-hoc networks (MANET) to provide digital communications that are robust to a turbulent operational environment. We also examine the state of software-defined networking (SDN). SDN is an emerging field that puts networking protocols on the same footing as ordinary computer programs. In theory, this should furnish computer networking with all the benefits of modern software engineering, such as reprogrammable hardware, rapid iteration, version control, and especially formal verification.

Section 6 describes a hypothetical application that might be used in a disruption-heavy network: a data replication service built on Yu and Vahdat's theory of "conits" (short for "consistency unit") [21]. This framework realizes a *continuous* consistency model in the sense that, as typically configured, it provides neither strong consistency nor guaranteed high-availability, but rather a quantifiable and controllable tradeoff between the two. The key idea is that many applications can tolerate inconsistency among replicas of a data item if an upper bound on the divergence between replicas is enforced. A conit-based database replication framework would allow system designers to define units of replicated data whose consistency is of interest, enforce policies bounding inconsistency between replicas of these items, and even dynamically tune these policies on the fly. We believe that only a conit-based replication infrastructure can provide the strict guarantees required for safety-related systems while also tolerating the adverse environments and real-world limitations of the systems we have in mind.

Section 7 concerns data fusion. Now and in the future, agents in disaster scenarios will make decisions informed by many different kinds of information. Efficient integration, processing, filtering, and dissemination of this information will be necessary to avoid "swimming in sensors and drowning in data" [8]. This task is especially challenging because agents will often work with incomplete or out of date information, and different sources of the same data may be contradictory, e.g. first responders may receive contradictory reports about whether a structure is occupied.

One promising trend in this space, which we briefly introduce in this section, is the development of sheaf theory as a natural mathematical model for data fusion [13]. Sheaf theory provides a rigorous framework for discussing how heterogeneous sources of noisy data can be integrated into a coherent picture, and can formally measure how well this task has been achieved.

We conclude in Section 8 by recapping some of the main themes in this document and highlighting areas where design decisions at various levels must be made to build a system that is tuned to the exact conditions we can expect from real-world scenarios. Such decisions might be informed by a combination of simulation and experimentation in the field.

2 Coordination Challenges in Disaster Response

This section explains aspects of disaster response scenarios that motivate the other sections of this document. We describe how real-world environments give rise to foundational challenges that must be addressed through the application of distributed computing principles. Even deploying the best communications equipment cannot realistically avoid the fundamental problems raised when many agents try to coordinate their actions over a widespread geographical area.

The operational environment of wildfire suppression, hurricane relief, and other disaster scenarios is generally characterized by systemic communications challenges. It is not surprising that a 2023 report on wildland firefighting modernization by the President's Council of Advisors on Science and Technology (PCAST) cites "the vulnerabilities and shortfalls in wildland firefighter communications, connectivity, and technology interoperability" in their first of five recommendations [11]. These shortfalls can be attributed to many factors inherent to disaster response: remote locations, difficult terrain, damaged infrastructure, harsh weather, and limited battery power, to name a few.

Agents in the field, such as first responders, generally experience high packet loss, high error rates, and unpredictable latencies when passing messages over the communication network(s). A conservative view suggests expecting the worst performance at particularly inopportune times. This is because conditions that prompt urgent communication can be expected to correlate with those that make communication difficult. For example, tautologically, a communications network is the most congested, and therefore the least available, precisely when everyone needs to use it. This phenomenon was famously exhibited in the immediate aftermath of the September 11^{th} terrorist attacks, where the sudden demand on multiple communications networks virtually brought them all to a halt (NEED CITATION).

From a systems perspective, an unreliable network presents a challenge for protocols used to coordinate distributed agents. Coordinated action requires enforcing some notion of consistency, i.e. agreement, across replicas of data shared between agents. We shall make this somewhat vague notion more precise in Section 3, but a simple example is that it is very important for everyone to agree which firetrucks have been dispatched to which scenes, or which radios have been reserved from the NIICD radio cache [10] and for whom. The exact meaning of "consistency" can

vary between applications. Generally, stronger consistency requirements are more difficult to maintain than relaxed ones because they require exchanging more information with other agents quickly, putting a heavier burden on the network. When the network is slow, system nodes (i.e. users or the devices they are using) that need to communicate may have to wait for network to deliver their messages, diminishing the efficacy of the system.

2.1 Communication and Safety

Many complications in the field are exacerbated by the basic fact that the network is never perfectly reliable. Faced with a poor communications environment, agents must choose between long delays in sending or receiving information, or acting with only limited knowledge. Typically they will experience some amount of both. Both options are problematic and present safety challenges because operational safety, by nature, requires agents to gather information about their environment and react to it. This information is received from distant sensors and other agents, who send it over the network(s), so poor network performance is a real problem.

Consider firefighting airtankers for example, the largest class of which are the Very Large Airtankers (VLATs), generally defined as those carrying more than 8,000 gallons of water or fire retardant (CITE). VLATs are commonly deployed to large wildfires in California, and the largest can deposit around 20,000 gallons of fire retardant, weighing about 170,000 pounds, in a single drop. Performed at a low altitude, a drop from a VLAT can easily crush a ground vehicle [12], posing a danger to firefighters. A 2018 accident led to the death of one firefighter and the injury of three others when an 87-foot Douglas Fir tree was knocked down by an unexpectedly forceful drop from a Boeing 747-400 Supertanker [2].

Suppose that in the future, firefighters are equipped with GPS sensors and digital transmitters—this could be an application on their personal cell phones, or something running on dedicated equipment. A seemingly reasonable policy would be to prohibit a VLAT from performing a drop if its computers do not have up-to-date information about the location of firefighters on the ground. The problem is that obtaining this information may be difficult or impossible if heavy smoke, a damaged radio tower, or a tall ridge prevents communications between the air and ground. In these scenarios, rigid enforcement of the safety policy would prevent airtankers from operating, making them ineffective.

This scenario exemplifies a classic tradeoff between opposing goals: system *safety* and system *availability* (or *liveness*), elaborated on in Section 3. In the distributed computing context, safety properties guarantee that a system will not perform an action that violates a constraint. In this example, a reasonable safety property could look like the following.

> $\mathbf{P}_{\text{safe}}$: All ground agents are known to be at least 100 ft.[2] outside the drop zone, and this information is current to within 30 seconds, or airtankers will not perform a drop.

[2]This figure comes from a recommendation (CITE).

By contrast, liveness properties stipulate that the system will certainly perform requested actions, typically within some time bound. In our scenario, an expected liveness property might be the following:

> $\mathbf{P}_{\text{live}}$: Airtankers will perform a drop within 20 minutes of receiving a request.[3]

Safety and liveness are frequently dual mandates: safety, in the sense used here, requires a system **never** to perform certain actions, while liveness requires a system to **always** perform certain actions. The tension between these ideals means the two often cannot be guaranteed simultaneously. Such is the case in our example: if firefighters are unable to broadcast their locations to the pilot, then the pilot's are impeded to maintain $\mathbf{P}_{\text{safe}}$ at the cost of $\mathbf{P}_{\text{live}}$, allowing the fire to spread in the meantime. [4]

Besides the safety/liveness tradeoff, the previous example exhibits two other aspects that are highly pertinent to reasoning about distributed systems. We pause to draw attention to them.

Epistemology Observe that the issue in the VLAT example does not simply disappear if no ground personnel are actually within 100 meters of a drop zone. That is, it is not simply a matter of whether a danger is actually present. To guarantee $\mathbf{P}_{\text{safe}}$, an airtanker's actions must be restricted when its computers do not *know* whether an action would violate $\mathbf{P}_{\text{safe}}$—knowledge of the fact, and not merely the fact of it, is the crucial part. In philosophical terms, the logic of distributed agents is inherently an *epistemic* one (CITE), meaning it must take into account not just what is true but what is known. The need to share knowledge is what drives communication and puts a burden on the network.

Discontinuity The properties $\mathbf{P}_{\text{safe}}$ and $\mathbf{P}_{\text{live}}$ are Boolean-valued, i.e. they are discrete, all-or-nothing propositions. The complexity of the operational environment demands considering more flexible kinds of properties. Suppose that agents are known to be 500 feet outside the drop zone, the extra margin meaning they are well away from any danger, but this information is only current to within 45 seconds. Clearly this is good enough information to authorize a drop, but strictly speaking it is a violation of $\mathbf{P}_{\text{safe}}$. To make the example more extreme, $\mathbf{P}_{\text{safe}}$ is violated even if the information is current to within 31 seconds, so a mere extra second of delay makes all the difference between whether the VLAT can operate, even though the firefighters in question are clearly far outside the drop zone. This is precisely the kind of *discontinuity* in system behavior that we aim to prevent, as it makes the system's behavior difficult to predict, being so sensitive to the particulars of the environment. Preventing this kind of discontinuity can be difficult because we still

[3] 20 minutes was cited as the response time within designated responsibility areas by the Chief of Flight Operations for Cal Fire https://www.youtube.com/watch?v=CWjfpMORGSQ

[4] A slight linguistic idiosyncrasy exhibited here is that liveness properties—not just "safety" properties—can also be relevant to human safety. Thus, the narrow technical meaning of safety properties for distributed systems does not capture the whole meaning of System Wide Safety.

want to provide rigorous guarantees about the system's behavior, i.e. we cannot forgo enforcing properties like $\mathbf{P}_{\text{safe}}$ altogether.

Figure 1: A DC-10 airtanker, rated for 9,400 gallons, drops retardant above Greer, Arizona.

2.2 Communication Patterns in the Field

Communication patterns in modern wildland firefighting are mostly simple, even primitive in comparison to what laymen often expect.

In the examples below, we note a kind of "geospatial locality of reference" that system designers should take into account. Generally speaking, we expect that agents with a higher need to coordinate their actions will tend to be located closer to each other, which in turn correlates with an ability to communicate quickly and reliably. This kind of principle motivates the sort of decentralized, ad-hoc networking protocols considered in Section 5. It can also affect the design of higher-level applications like the one in Section 6.

Communication on the ground In the field, communication between firefighters and other agents in disaster response scenarios is often facilitated by handheld

Figure 2: The Ironside Mountain lookout station, destroyed in the 2021 Monument fire, shown with protective foil on August 10$^{\text{th}}$, 2015 during the 2015 River Complex fire. This particular fire burned 77,077 acres over 77 days.

(analog) radios, which are inherently limited in their battery life,[5] 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 to communicate using the simplest of technologies: shouting back and forth. Besides highlighting the fact that sometimes simple things work, this is a clear manifestation of geolocality of reference: firefighters mostly need to communicate when they are working near each other, in some cases so nearby they can communicate without network infrastructure.

Communicating over a long distance typically requires infrastructural support, such as the use of cell towers and repeater stations. Typically, disaster response environments have scarce permanent infrastructure: perhaps a few repeaters mounted to a watch tower for a wildland fire setting. Ad-hoc infrastructure, such as a cell on wheels (COW) or cell on light truck (COLT), can sometimes be deployed on an as-needed basis if the location allows for it. A common issue is making sure that all equipment is properly configured, for instance that all radios are listening on the correct frequencies, particularly when different agencies and groups need to interoperate.

Use of centralized infrastructure comes at the cost of potential widespread failure when the infrastructure fails. For example, in California, the Ironside Mountain lookout/repeater station was destroyed during the 2021 Monument Fire,[6] which

[5] A coordinator for NIFS reported that during peak demand, wildland firefighters in the United States go through upwards of a combined 350,000 "AA" batteries in a day.

[6] https://www.fire.ca.gov/incidents/2021/7/30/monument-fire/

burned approximately 184,142 acres over 88 days. 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 (CITE), 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. This is exactly the kind of scenario considered by Brewer's CAP theorem in Section 3.

> 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.

Vehicles on the ground Large numbers of ground vehicles are involved in wildfire suppression. A large wildfire response can involve more than 50–100 firetrucks distributed over a large geographical area. 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, which is to say better, communications equipment than a human. For instance, a vehicle could be equipped with a satellite link as well as a local wireless area network (WLAN) base station, serving as a bridge between agents in the field and central coordinators (e.g. incident commanders).

Communication in the air Wildland firefighting increasingly involves the use of helicopters and fixed wing aircraft, but 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 a half dozen or more, may need to operate in a small area, near complex terrain, during adverse conditions, often at a low altitude.[7] In other words, the demands are many and the margins for error are small. This sort of use case requires 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-site to ground crews than ground crews have to each other, firefighters sometimes relay messages to air-based units over radio, which in turn is relayed back down to other ground units. The locality principle comes into play here too, but this time in the reverse direction: this relay

[7] Current recommendations (WHOSE) call for VLATs to perform drops at 250 ft. above the tree canopy, while smaller aircraft may go lower.

scheme allows knowledge to travel farther, but the extended reach comes at the cost of introducing delays and possible degradation of message quality, as in the classic game of telephone.

In some cases, planes from the Civil Air Patrol[8] (CITE) have been equipped with radio repeaters and dispatched to wildfires to provide service to ground-based units. In the future, this sort of service could be provided, perhaps autonomously, by base stations mounted to unmanned aerial vehicles (UAVs), which might perform additional functions such as tracking the fire perimeter. More generally, a future communication system should exploit every pass messages between agents, even in the event of infrastructure failure, in a transparent and decentralized fashion. For instance, firefighters may use an ad-hoc network built from handheld devices, vehicle-mounted devices, temporary structure, an devices attached to overhead aircraft. In effect, this would be a high-tech modernization of the sort of informal relay schemes operating today over traditional radio channels.

2.3 Towards the Future

- CITE developed a prototype application for firefighters in the field.
- Talk about other applications

Taking a broader view, agents in disaster response environments will often be both producers and consumers of data, and this data will need to processed for agents to make informed decisions. Our background research indicated many different kinds of data that could be valuable for responders. To give just a glimpse:

- Free-form communication, especially recorded voice messages to be broadcast to many firefighters at once.
- The exact or estimated location of firefighters, vehicles, victims, hazards, etc.
- Medical information, perhaps collected from digital triage tags (CITE)
- Data about current and predicted fire or weather patterns
- Topographic information about the terrain, highlighting for instance the location of rivers and roads that could form a fire control line
- Planned escape routes, safety zones, and landing zones
- Availability and dispatching of resources, e.g. ambulances, airtankers, or crews on standby

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—all problems that are further exacerbated by an unreliable network.

For instance, a central data fusion center may be used to detect and alert responders to a fire that has accidentally moved beyond a control line (known as *slopover*). Such information would be of high importance, and it would be worthwhile to expend network resources conveying this information to the relevant parties. On the other hand, a firefighter may not need the minute GPS position of every other

[8] A civil auxiliary of the U.S. Air Force. https://www.gocivilairpatrol.com/

firefighter in the field: perhaps the exact location of teammates, and the general location of other crews.

Warn firefighters when they are too far from an escape route or safety zone.

For developers, it is clear that this requires some coordination between the application and the networking layers, as the latter must be given enough information to make the most prudent use of limited resources.

3 Introduction to Distributed Systems

In this section we attempt to summarize the main themes of distributed systems theory and where this fits into the picture painted in Section 2. The primary reference we are following is the excellent book by Kshemkalyani and Singhal [7], particularly chapters 1, 3, 6, and 12.

A distributed system, in the most general sense, is a collection of independent entities that cooperate to solve a problem that cannot be individually solved [7]. This memo is concerned with distributed systems used by firefighters and other disaster response agents. The components of our system could be computers, smartphones, various types of sensors, or other kinds of communication and networking equipment. These may be personal devices carried by individuals, mounted to ground or air-based vehicles, or be deployed as temporary or permanent infrastructure. The clients of these devices could be civilians, firefighters, incident commanders, or other agents involved in disaster response efforts. The devices could also be autonomous. The kinds of tasks our system or its components might handle include navigating safely in close proximity, delivering resources to remote locations, suppressing fires, and collecting, processing, and disseminating data about the environment.

3.1 Message Passing and Logical Time

Singhal and Shivaratri [14] offer the following definition of a distributed computing system:

> "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."

We shall consider a system consisting of a set $\mathcal{P} = \{P_i\}_{i \in I}$ of *processes* executing on independent, probably geographically dispersed computing and communication devices. We shall consolidate all these kinds of devices under the name "nodes." It is important to note that different kinds of system nodes can have different characteristics, such as capabilities and intended uses, but for now we take a high-level view.

A notable aspect of what makes the system "distributed" is that the nodes must communicate by message-passing, which must be facilitated by some kind of network. This restriction is implied by absence of a common memory (whereas

processes running on the same machine can share information by writing it to a common memory location).

A foundational assumption is that the network is unreliable. Section 2 explained why this is certainly expected in disaster scenarios. Delivering messages over the network is not instantaneous—message packets must be sent over the air or through a cable, and typically processed and forwarded by any number of intermediate networking devices, all of which takes time. This means that messages are passed with some unpredictable latency. In this section, our main challenge is to provide ways for agents to coordinate their actions despite the uncontrollable delays in communication caused by the network and environment, which is to say by the "distributed" aspect of the system.

Network reliability In general, the latency involved in sending message need not even be finite, meaning the network may not deliver a packet at all, i.e. it may silently discard the packet without alerting the sender. The same message may be also be delivered multiple times, and different messages may arrive in any order. Information corruption, e.g. flipping a 1 to a 0, is also expected for some fraction of total transmissions. In the case of *Byzantine faults* (CITE) the network might even act like a malicious adversary. In this section, being focused on higher-level considerations, we shall assume that messages are delivered exactly once, uncorrupted, but with an unpredictable delay. Some, but not all, common so-called "transport" protocols like TCP (CITE) and LTP (CITE) can provide this sort of semi-reliable message delivery over a less reliable network. For example, these protocols might tag messages with unique identifiers that allow the receiver to detect duplicates and arrange messages in the ordered intended by the sender. For now, one could say we are working at a level of abstraction above the network transport layer. (We shall take a closer look at low-level networking details in Section **??**, where we relax these assumptions.) Note that these reliability mechanisms contribute to the latencies involved in message passing, so they do not solve the problems under consideration here, namely the difficulty of coordination when messages are unpredictably delayed and without access to sychronized clocks.

3.1.1 FIFO and causal ordering

Figures 3 and 4 show time diagrams for messages $\{m_i\}_{i=1}^{N}$ sent between three processes P_1, P_2, and P_3. The x-axis of the diagram shows the flow of physical time from left to right. Each process consists of a worldline depicting the events occurring in that process. Each message m_i starts off as a message *send* event m_i^{send} indicating the moment the message is sent over the network by its originating process; its delivery generates a message *receive* event m_i^{recv}. The corresponding events are connected by an arrow. The subscripts on the messages only serve to disambiguate them, but they are not part of the message contents and have no semantic importance.

Two things are clear from these figures. First, messages are sent with a varying delay. This delay is why the arrows are diagonal rather than perfectly vertical. Second, messages m_i and m_j sent by different senders, whether to the same recipient

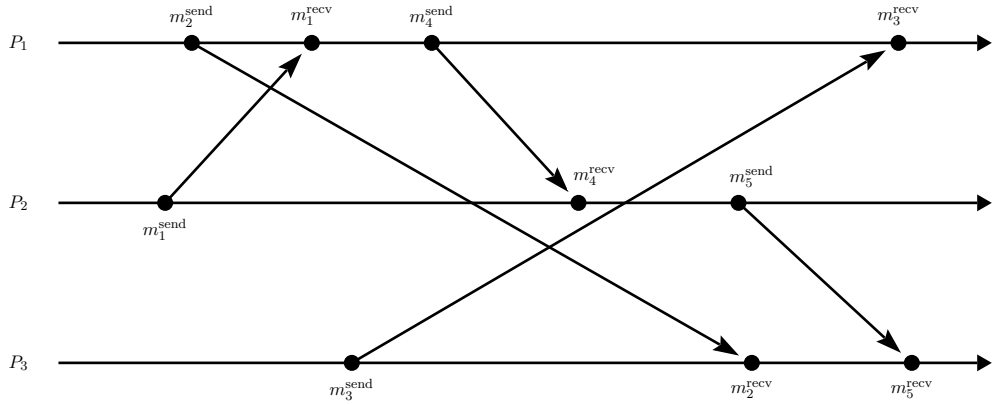


Figure 3: Time diagram depicting latencies in message-passing

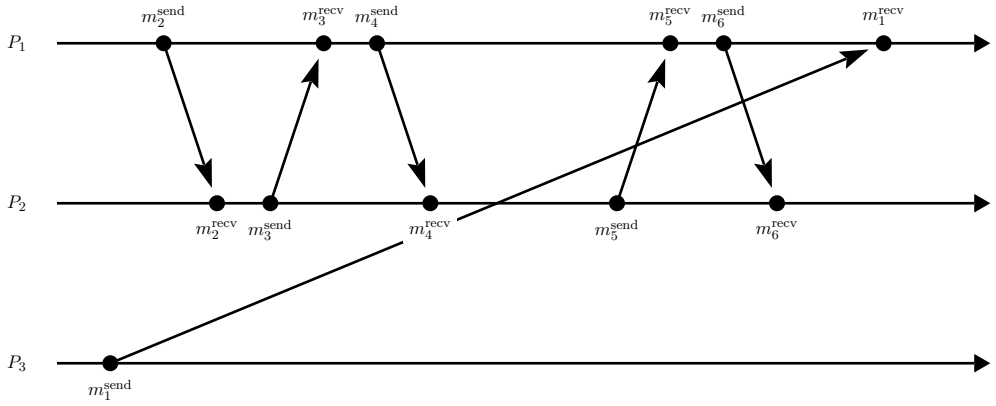


Figure 4: Message m_1 is sent first and delivered last

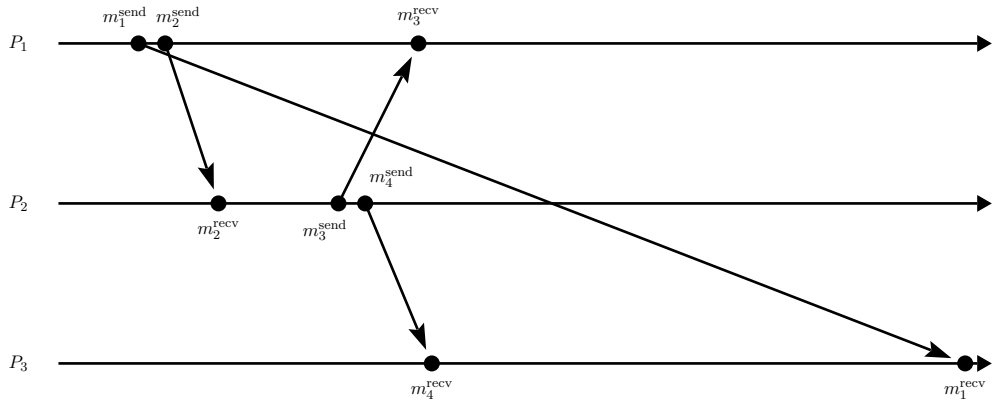


Figure 5: A time diagram satisfying FIFO but violating causal ordering

or different ones, may arrive in different order than they were sent in. Such is the case with messages m_2 and m_4 in Figure 3, sent by P_1 to P_3 and P_2, respectively. This diagram may depict a scenario where P_1 and P_3 are far apart and have a bad connection to each other, with P_2 having a relatively good connection to the others. In Figure 4, m_1 is sent before all other messages, but it is last to be delivered, with all of the messages sent by P_2 delivered earlier despite being sent later. This may correspond to a scenario where P_2 has moved closer to P_1 while P_3 has moved further away.

At this level of abstraction it is reasonable to assume that messages from the same sender to the same recipient arrive in the correct order, an assumption known as the FIFO (first-in, first-out) network model (CITE). A violation of the FIFO assumption would correspond to two crossing lines that have *the same sender-receiver pair*. All figures in this section satisfy FIFO, as no two intersecting lines are originate at the same process and end at the same receiver. FIFO ordering is one of the basic services that might be provided by the network transport protocol.

Unfortunately, things can go wrong for distributed applications even assuming the FIFO networking model. Namely, messages can violate *causal ordering* (CO). We define CO formally before giving a more informal description.

Definition 3.1 (Causal ordering)**.** First we define a binary relation $\rightarrow$ of "direct causality" as follows:

$$e \rightarrow e' \iff \begin{cases} e \leq_{P_i} e' \text{ for some process } P_i \\ e = m_i^{\text{send}} \text{ and } e' = m_i^{\text{recv}} \end{cases}$$

That is, event e directly causes e' if the two events occur on the same process and e happens before e', or e is a message send event and e' is the corresponding message receive event. Now we define a causality relation recursively as follows:

$$e \rightarrow e' \iff \begin{cases} e \rightarrow e' \\ \exists e'' \text{ such that } e \rightarrow e'' \text{ and } e'' \rightarrow e \end{cases}$$

That is, event e causes e' if e directly influencs e' or there is some intermediate event e'' such that e causes e' and e' causes e''. The reader may note that $\rightarrow$ can concisely be defined as the transitive closure $\rightarrow^*$ of $\rightarrow$.

Informally, $e \rightarrow e'$ holds whenever one can put a finger on an event e in the time diagram and, finger held down, trace a path to e' just by moving left-to-right on the same worldline OR following a diagonal lines connecting the sending and receiving events of a message. Calling this "causal" ordering can be a misnomer, however, as it may be that $e \rightarrow e'$ without it being the case that e "caused" e'. Intuitively, it would be more pedantic (but more wordy) to say that information from e potentially affected, or could have influenced, the event e'. For example, if e corresponds to P receiving a message, and e' corresponds to P sending a message, then it is possible that the message in e' contains information that was learned from e.

Let us examine a simple application of causal ordering and why a FIFO model may violate it. Consider a naïvely-designed text-messaging application used by three

clients P_1, P_2, P_3. To post a message, a process simply sends the contents of that message to the other two members of the group. The obvious way to do this is to to send two messages, one to both other processes. (Note that we do not assume the ability to send one network message to multiple recipients at once, since that sort of multicast mechanism is essentially what we are trying to implement.) For example, perhaps P_1 posts a question "Is the house at the end of the street occupied?" P_2 responds to this question in the affirmative. Figure 5 depicts a possible diagram showing these four messages (two messages each for the question and the answer). The diagram vaccuously satisfies FIFO because all four message send events have a distinct sender-receiver pair (recall that FIFO requires distinct messages with the same sender-receiver combination to arrive in the order they were sent in). However, P_3 witnesses P_2's answer to P_1's question before seeing the question posed by P_1 in the first place. In this simple example, P_3 may be able to infer what is meant, but clearly in a more complex example with potentially hundreds of clients, the potential for inconsistency and serious confusion is great when causal ordering is violated. Some mechanism for tracking causal precedence is clearly needed.

At first it may appear that we can avoid the scenario described above by attaching to each event a *timestamp* tracking when the event occurred—this would seemingly allow P_3 to infer that the message sent by P_1 logically precedes P_2's response. Let us see why this fails. For each event e, let $C(e)$ denote the timestamp attached to that event. The fundamental *monotonicity property* we want to satisfy is the following:

$$\text{for all events } e \text{ and } e',\ e \to e' \implies C(e) < C(e') \tag{MP}$$

MP requires that if e causally precedes e', then e should have a lesser timestamp. In our example, with $m_1^{\text{send}} \to m_4^{\text{send}}$, it would have to be the case that $C(m_1^{\text{send}}) < C(m_4^{\text{send}})$. The problem is that this cannot be guaranteed unless P_1 and P_2 have synchronized clocks, and a hallmark of distributed systems is we cannot make this sort of assumption (as stated in the quote at the beginning of this section). Physical clocks suffer from drift, which is to say they do not all run at the same physical rate. They are also quite prone to misconfiguration, say if a device administrator sets the time or date incorrectly, and they require an always-on battery to track time when the device is powered off, say when sitting in a storage. While in some cases one may assume access to clocks that are only approximately synchronized, perhaps by employing special-purpose protocols like NTP (CITE), in general we cannot assume access to physical clocks that are so precise as to fundamentally solve the problems discussed in this section. It is also not the case that merely timestamping messages is enough to answer every question we could ask about causal precedence. For example, it is especially desirable if we can assume the *strong clock consistency condition*:

$$\text{for all events } e \text{ and } e',\ e \to e' \iff C(e) < C(e') \tag{SC}$$

That is, SC strengthens MP by requiring that we can always compare two timestamps to infer whether the events are causally related. This property cannot hold even with synchronized clocks, as events need not be causally related even if one physically precedes the other. For instance, in Figure 4, P_3's message m_1 is sent

before any other messages, but m_1^{send} does not causally precede any event except m_1^{recv}. To solve these sorts of problems, we turn our attention to *logical* clocks.

3.1.2 Logical clocks

Distributed applications can systematically keep track of causal precedence by employing a system of logical clocks, which typically measure logical (non-physical) time in terms of integers or sets of integers. Each device maintains a logical clock that is advanced according to certain rules. Three major kinds of "logical clocks" are commonly considered: scalar clocks, vector clocks, and matrix clocks. Scalar clocks are extremely simple but provide the most coarse-grained information, while matrix clocks require storage space quadratic in the number of processes but provide extremely precise information about causality; vector clocks fall somewhere in the middle. We provide a brief description of each paradigm, with the obligatory note that variants of these schemes also exist.

Scalar clocks Scalar clocks, also called Lamport clocks after their inventor (CITE), require each process P_i to maintain a single scalar value C_i, a non-negative integer, which we assume is initialized two 1. There are two update rules:

1. When a message is sent, C_i is updated according to the rule

$$C_i := C_i + 1.$$

   This new value is the timestamp attached to the message.

2. When a message is received with timestamp C, C_i is updated according to

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

   This new value is the timestamp attached to the message receive event.

Figures (REF–REF) depict Figures (REF–REF), respectively, this time showing the scalar timestamp that would be assigned to each event.

It is not hard to see that scalar clocks satisfy the monotonicity property MP. Examining Figures 4 and 7, we see that P_3 could examine timestamps to determine that $C(m_1^{\text{send}})$ (the timestamp sent with m_1, value 1) is less than $C(m_b 4^{\text{send}})$ (value 5). Therefore the application should make it clear to the client to m_1 logically precedes the response from P_2, even if it arrives later, resolving the original problem.

However, scalar clocks do not satisfy the strong monotonicity condition. If a process sees two messages m and m' sent by P and P', respectively, we cannot examine timestamps to infer whether P' message contents could have been causally influenced by the contents of m. For example, examining Figures 5 and 8, we see that $C(m_1^{\text{send}})$ has a globally minimal timestamp value of 1. However, m_1 does not logically precede any other message. For this fine-grained information we must use vector clocks.

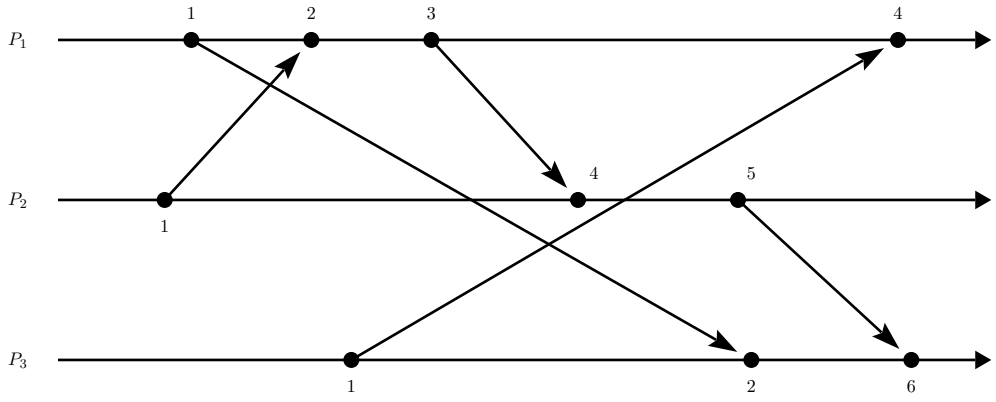


Figure 6: Scalar clocks for Figure 3

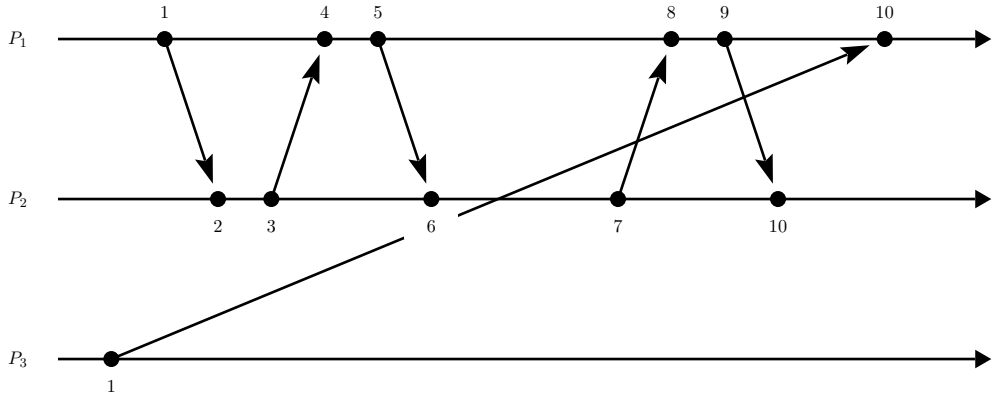


Figure 7: Scalar clocks for Figure 4

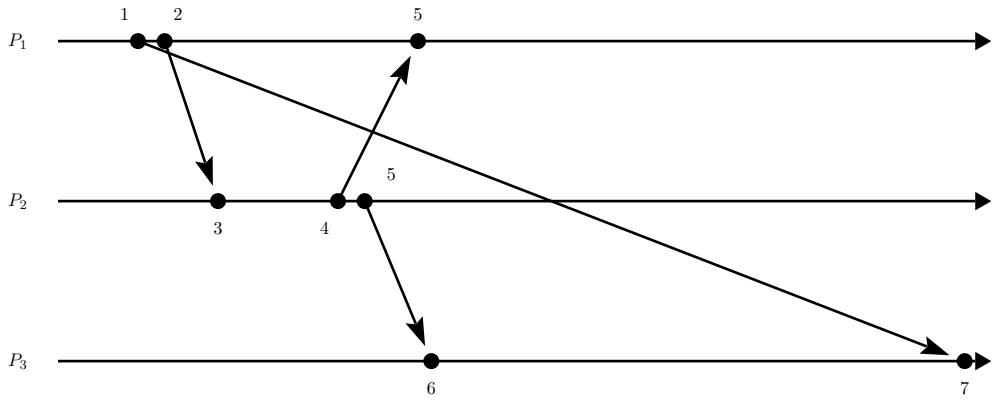


Figure 8: Scalar clocks for Figure 5

Vector clocks Vector clocks require each process to store a scalar value for each process in the system. That is, given a set of N processes, P_i maintains a vector $\mathrm{vt}_i[1 \ldots N]$ of non-negative integers, with all values initialized to $\mathrm{vt}_i[x] = 1$. As with scalar clocks there are two update rules:

1. When a message is sent, vt_i is updated according to the rule

$$\mathrm{vt}_i[i] := \mathrm{vt}_i[i] + 1.$$

   The entire (updated) vector vt_i is attached to the message.

2. When a message is received with timestamp vt, vt_i is updated according to

$$\mathrm{vt}_i[x] := \max(\mathrm{vt}[x], \mathrm{vt}_i[x]) \quad \text{for all } x = 1 \ldots N.$$

   Then P_i advances its own local time according to the rule

$$\mathrm{vt}_i[i] := \mathrm{vt}_i[i] + 1$$

   This new vector is the timestamp attached to the message receive event.

A process P_i' vector clock can be concisely described as an ordinary scalar clock $\mathrm{vt}_i[i]$ combined with, for all $j \neq i$, a *lower bound estimate* $\mathrm{vt}_i[j]$ of the local scalar clock $\mathrm{vt}_j[j]$ of process P_j.

Figures (REF–REF) depict Figures (REF–REF), respectively, this time showing the vector timestamp that would be assigned to each event.

Let us see an example of where the extra information provided by vector clocks is useful.

- Is the structure occupied?
- There are still two people trapped on the second floor.
- We already sent a team in and they determined it was empty.

The order of the last two messages is important.

Matrix clocks Matrix clocks are to vector clocks what vector clocks are to scalar clocks. That is, a matrix clock is a vector clock combined with a lower bound estimate of every other process's vector clock.

It is worth highlighting the fact that matrix clocks are used to keep track of (a lower bound estimate of) what other processes know about the global state of the system. In Section 3, we explained that distributed systems fundamentally involve epistemic logic, i.e. reasoning about not just what is true but what is known. In Section 6 we will see an example where this sort of lower bound estimate of other process's clocks is important for distributed database replication.

3.2 Shared Memory

A fundamental distributed application is the *shared distributed memory* abstraction. We shall assume that all processes maintain a local replica of a globally shared data object, as replication increases system fault tolerance. For simplicitly, we shall discuss the data store as a simple key-value store, but it could be something else like a database, filesystem, persistent object, etc.

3.2.1 Client-server model

Process take requests from clients, such as to read or write a value in a database. The lifecycle of a typical request is depicted in Figure 9. At some physical time (i.e. wall-clock time) $C.s \in \mathbb{R}$ (client start time), a client sends a message to a process. At time $E.s$, which we'll call the *start time* of the event, the message is accepted by the process. The request is processed until some strictly greater time $E.t > E.s$ when a response is sent back to the client. The value $E.t - E.s$ is the *duration* of the event.

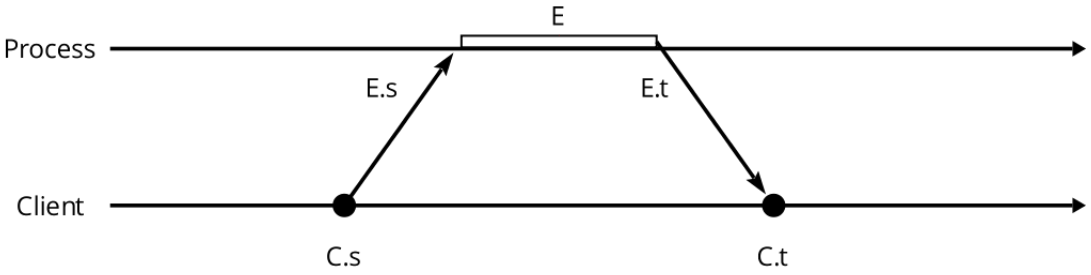


Figure 9: Lifetime of a client request

While handling the request, the process may coordinate with other processes in the background by sending and receiving more messages. For example, the process may propagate the client's request to other processes, retrieve up-to-date values from other processes to give to the client, or delay handling the client's request in order to handle other requests.

We assume clients submit two types of requests to processes. A *read request* is a request to lookup the current value of a variable. A request to read the variable x that returns value a is written $R(x, a)$. A *write request* is a request to set the current value of a variable. Notation $W(x, a)$ represents writing value a to x. We assume all processes provide access to the same set of shared variables.

3.2.2 Causal precedence/external order

We shall consider the full set of events across a distributed system, such as shown in Figure 10. This is called an *execution*. Consistency models constrain the set of allowable return values in response to clients' requests.

We shall assume that requests handled by a single process do not overlap in time. This can be enforced with local serialization methods such as two-phase

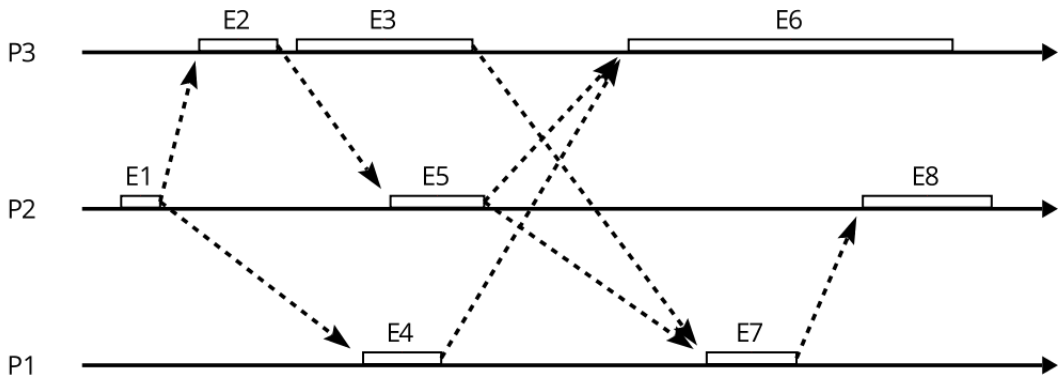


Figure 10: Depiction of external order between concurrent events across three processes. Intra-process and transitive edges are not depicted.

locking (CITE) that can be used to isolate concurrent transactions from each other, providing the abstraction of a system that handles requests one at a time. On the other hand, any two processes may handle two events at the same physical time, so that there is no obvious total order of events across the system. Instead, one has a partial order called *external order*. Intuitively, it is the partial order of events that would be witnessed by an observer recording the real time at which systems begin and finish responding to requests.

Definition 3.2. Let E be an execution. Request $E1$ *externally precedes* request $E2$ if $E1.t < E2.s$. That is, if the first request terminates before the second request is accepted. This induces an irreflexive partial order called *external order*.

Recall that an irreflexive partial order is a binary relation $<$ such that $A \not< A$, $A < B \implies B \not< A$, and $A < B, B < C \implies A < C$.

Because we assume processes handle events one-at-a-time, the events handled at any one process are totally ordered by external order—one event cannot start before another has finished. If $E1$ and $E2$ are events at *different* processes, they need not be comparable by external order, i.e. neither $E1.t < E2.s$ nor $E2.t < E1.s$, making them *physically concurrent*.

Definition 3.3. If two events overlap in physical time (equivalently, if they are not comparable by external order), we call the events *physically concurrent* and write $E1||E2$.

Physical concurrency is a reflexive and symmetric—but usually not transitive—binary relation. Such structures are often called *compatibility relations.* The general intuition is that anything is compatabile with itself (reflexivity), and the compatibility of two objects does not depend on their order (symmetry). But if A and B are compatible with C, it need not be the case that A and B are compatible with each other.

Figure 10 shows the external order relation for an execution. To save space we elide arrows between two events of the same process and arrows that can be inferred

by transitivity. This corresponds to the directed acyclic graph structure shown in 11.

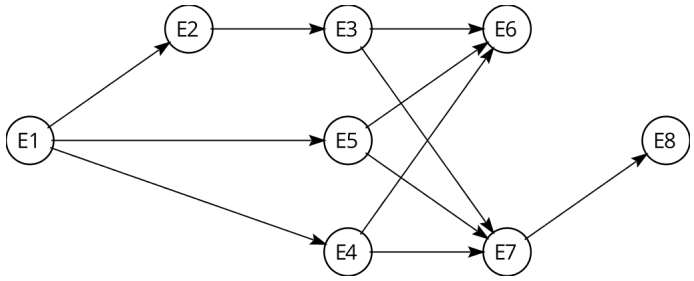


Figure 11: The directed acyclic graph (DAG) induced by external order.

The reader may wonder if we can consider events to be totally ordered, say by pairing them with a timestamp that records their physical start time to resolve ties like $x||y$. This order is not generally useful for a couple reasons. First, we assume processes have only loosely synchronized clocks, so timestamps from two different processes may not be comparable. Additionally, even systems that enforce linearizable consistency (c.f. Section **??**) do not necessarily handle requests in order of their physical start times.

3.3 Consistency Models

A fundamental goal for distributed computing systems is to "[appear] to the users of the system as a single coherent computer" [15]. Enforcing consistency means clients are presented with the abstraction of a shared world, as if they are all connected to a central computer rather than than the complex system of loosely coupled computers they are actually using.

Considering the shared memory abstraction, the requirement is that all nodes present a *mutually-consistent* view of the shared memory to system clients. That is, as far as clients are concerned, it should appear there is a single source of truth that all system nodes have instantaneous access to. Bad things can happen when consistency is violated:

- If two air traffic controllers were presented with conflicting information about the trajectory of aircraft, they could potentially issue dangerously incongruous instructions to pilots.
- Messages between users that are delivered in the wrong order can lead to misunderstandings. For instance, it may be appear to Bob that Alice is responding to a different question than the one she meant to if her messages arrive in the wrong order.
- A bank client would be unhappy if deposits that appear in their account online are not reflected when they check their balance at an ATM, or if they seem to disappear after refreshing the webpage.

- Resource-dispatching systems are not useful if a resource that appears to be available cannot be used because the information is out of date. Conversely, inefficiencies result if clients think available resources are actually unavailable.

Strongly consistent systems are easier to understand and reason about. Violating consistency means the abstraction of a shared universe is broken, which can invalidate clients' mental model of the system, make the system's behavior harder to predict, or cause safety requirements to be violated. Clearly, all other things being equal, one wants distributed systems to have as much consistency as possible. However, we shall see that strong notions of consistency are brittle in the sense that they generally cannot be achieved for the kinds of systems we consider in this document.

What precisely constitutes consistency? A *memory consistency model* formally constrains the allowable system responses during executions. "Strong" models are generally understood as ones providing the illusion that all clients are accessing a single global database. As we will see, this still leaves room for different possible behaviors (i.e. allows non-determinism in the execution of a distributed application), but the allowable behavior is tightly constrained. Weaker models tolerate deviation from this standard. For a given application, the most appropriate model depends on the semantics expected by the application and its clients, which must be weighed against other requirements.

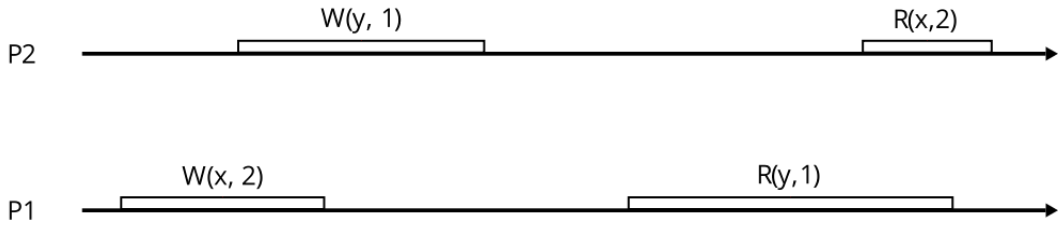
3.3.1 Linearizability/atomic consistency

Linearizability (Herlihy and Wing 1990) is essentially the strongest common consistency model. It is known variously as atomic consistency, strict consistency, and sometimes external consistency. In the context of database transactions (which come with other guarantees, like isolation, that are more specific to databases), the analogous condition is called strict serializability. A linearizable execution is defined by three features:

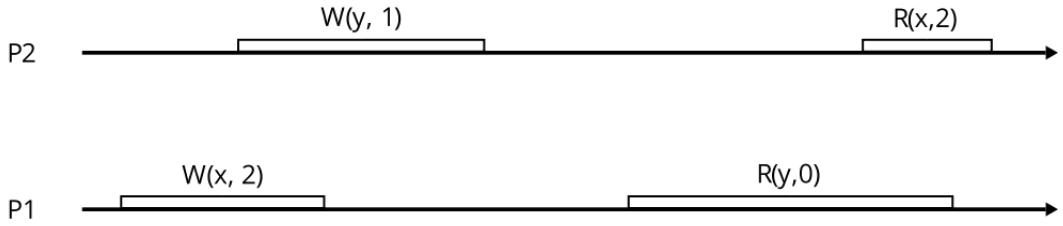
- All processes act like they agree on a single, global total order defined across all accesses.
- This sequential order is consistent with the actual external order.
- Responses are semantically correct, meaning a read request $R(x, a)$ returns the value of the most recent write request $W(x, a)$ to x.

Figure 12a shows a prototypical example of a linearizable execution. We assume that all memory locations are initialized to 0 at the system start time. Intuitively, it should appear to an external observer that each access instantaneously took effect at some point between its start and end time. Hence, the request to read the value of y returns 1, because at some point between $W(y, 1).s$ and $W(y, 1).t$ that change took effect. If client on P_1 read a stale value, we would say the execution is not linearizable. Figure 12b shows an non-linearizable execution that returns stale data instead of reflecting the write access to y on $P2$.

Linearizability can be precisely defined in terms of *linearizations.*



(a) A linearizable execution. Any choice of linearization works here.



(b) A non-linearizable execution. The request to read y returns a stale value.

Figure 12: A linearizable and non-linearizable execution.

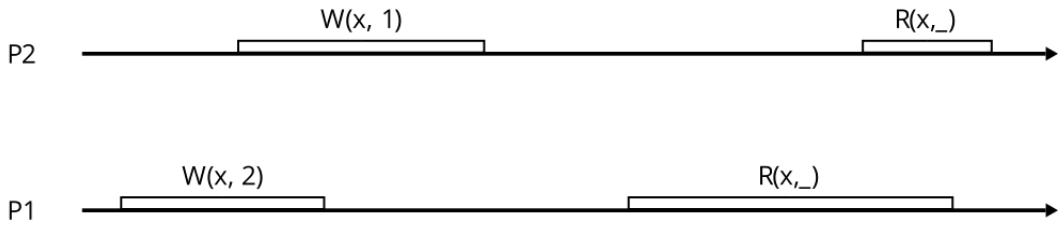
Definition 3.4. A *linearization point* $t \in \mathbb{R} \in [E.s, E.t]$ for an event E is a time between the event's start and termination. An execution is *linearizable* if and only if there is a choice of linearization point for each access, which induces a total order called a *linearization*, such that E is equivalent to the serial execution of events when totally ordered by their linearization points.

Linearization points are demonstrated in Figure 13. The figure shows different linearizable behaviors in response to the same underlying set of accesses. It is assumed no distinct access can have the same linearization point, so that we get a total order. This demonstrates that linearizability still leaves some room for non-determinism in the execution of distributed applications. In this example, the requests must both return 1 or 2. The constraint is that the values must agree—linearizability forbids the situation in which one client reads 1 and another reads 2 (Figure 14).

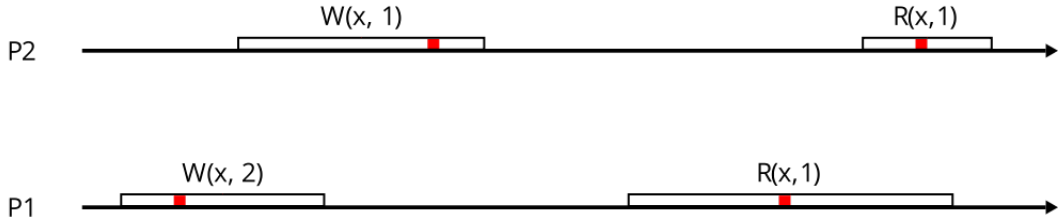
Definition 3.5. We say an entire system is linearizable when all possible executions of the system are linearizable.

3.3.2 Sequential consistency

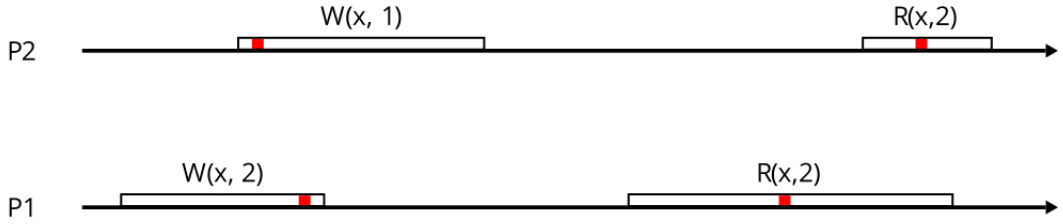
Enforcing atomic consistency means that an access E at process P_i cannot return to the client until every other process has been informed about E. For many applications this is an unacceptably high penalty. A weaker model that is still strong enough for most purposes is *sequential* consistency. This is an appropriate model if a form of strong consistency is required, but the system is agnostic about the precise physical time at which events start and finish, provided they occur in a globally agreed upon order.



(a) An execution with read responses left unspecified.



(b) A linearizable execution for which both reads return 1.



(c) A linearizable execution for which both reads return 2.

Figure 13: Two linearizable executions of the same underlying events that return different responses. Possible linearization points are shown in red.

A sequentially consistent system ensures that any execution is equivalent to some global serial execution, even if this is serial order is not the one suggested by the real-time ordering of events. When real-time constraints are not important, this provides essentially the same benefits as linearizability. For example, it allows programmers to reason about concurrent executions of programs because the result is always guaranteed to represent some possible interleaving of instructions, never allowing instructions from one program to execute out of order.

Definition 3.6. A *sequentially consistent* execution is characterized by three features:

- All processes act like they agree on a single, global total order defined across all accesses.
- This sequential order is consistent with the program order of each process.
- Responses are semantically correct, meaning reads return the most recent writes (as determined by the global order)

Processes in a sequentially consistent system are required to agree on a total order of events, presenting the illusion of a shared database from an application programmer's point of view. However, this order need not be given by external order. Instead, the only requirement is that sequential history must agree with process order, i.e. the events from each process must occur in the same order as in they do in the process. This is nearly the definition of linearizability, except that external order has been replaced with merely program order. We immediately get the following lemma.

Lemma 3.1. *A linearizable execution is sequentially consistent.*

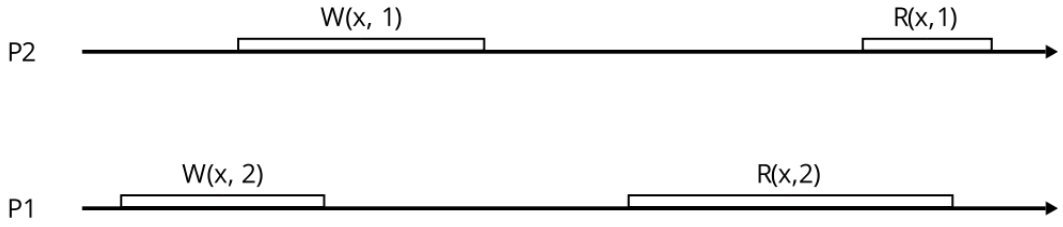
Proof. This follows because process order is a subset of external order. □

Visually, sequential consistency allows reordering an execution by sliding events along each process' time axis like beads along a string. Two events from the same process cannot pass over each other as this would violate program order, but events on different processes may be commuted past each other, violating external order. This sliding allows defining an arbitrary interleaving of events, a totally ordered execution with no events overlapping. From this perspective, while linearizability requires the existence of a linearization, sequential consistency requires the existence of an equivalent interleaving.

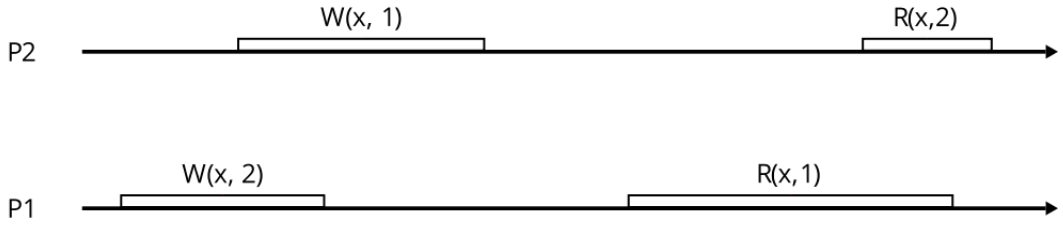
The converse of Lemma 3.1 does not hold. For example, Figure 15a was previously shown (Figure 14a) as a nonlinearizable execution. However, it is sequentially consistent, as evidenced by the interleaving in Figure 15b that slides the events $W(x, 1)$ and $R(x, 2)$ past each other.

3.3.3 Causal consistency

Causal consistency (CITE) is a weak (i.e. non-strong) consistency model

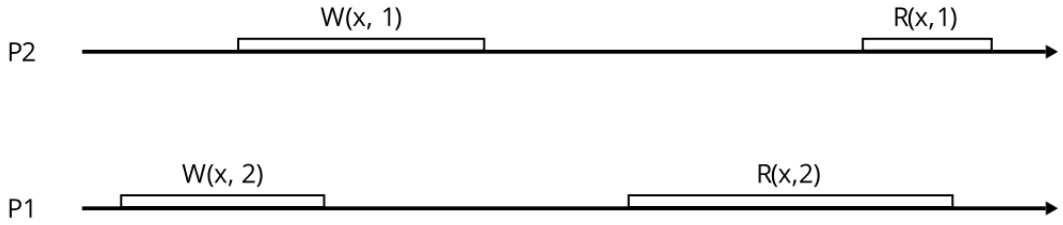


(a) A nonlinearizable execution with the read access returning disagreeing values. We will see later (Figure 15) that this execution is still sequentially consistent.

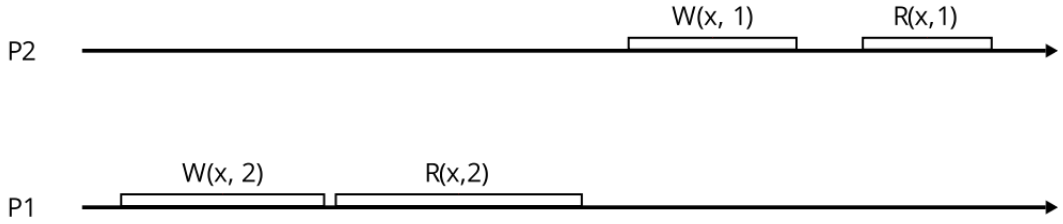


(b) Another nonlinearizable execution with read access values swapped. This execution is not sequentially consistent.

Figure 14: Two non-linearizable executions of the same events shown in Figure 13.

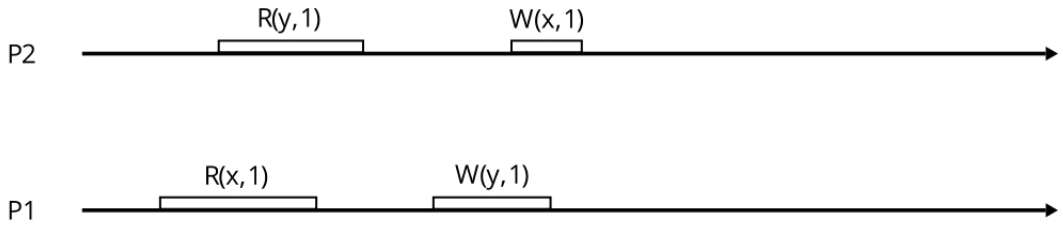


(a) A non-linearizable, sequentially consistent execution.

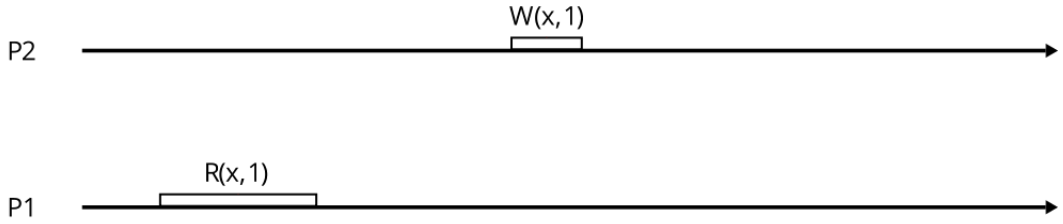


(b) An equivalent interleaving of 15a.

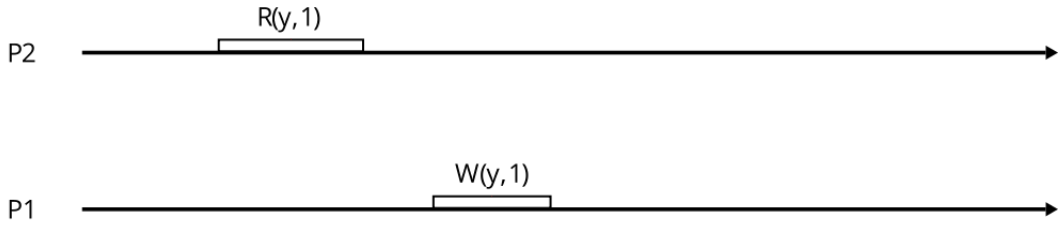
Figure 15: A sequentially consistent execution and a possible interleaving.



(a) A non-sequentially consistent execution.



(b) The sequentially consistent history of x.



(c) The sequentially consistent history of y.

Figure 16: A non-sequentially consistent execution with sequentially-consistent executions at each variable.

3.4 The CAP Theorem

Real-world systems often fall short of behaving as a single perfectly coherent system. The root of this phenomenon is a deep and well-understood tradeoff between system coherence and performance. Enforcing consistency comes at the cost of additional communications, and communications impose overheads, often unpredictable ones.

Fox and Brewer [4] are crediting with observing a particular tension between the three competing goals of consistency, availability, and partition-tolerance. This tradeoff was precisely stated and proved in 2002 by Gilbert and Lynch [5]. The theorem is often somewhat misunderstood, as we discuss, so it is worth clarifying the terms used.

Consistency Gilbert and Lynch define a consistency system as one whose executions are always linearizable.

Availability A CAP-available system is one that will definitely respond to every client request at some point.

Partition tolerance A partition-tolerant system continues to function, and ensure whatever guarantees it is meant to provide, in the face of arbitrary partitions in the network (i.e., an inability for some nodes to communicate with others). It is possible that a partition never recovers, say if a critical communications cable is permanently severed.

A partition-tolerant CAP-available system cannot indefinitely suspend handling a request to wait for network activity like receiving a message. In the event of a partition that never recovers, this would mean the process could wait indefinitely for the partition to heal, violating availability. On the other hand, a CAP-consistent system is not allowed to return anything but the most up-to-date value in response to client requests. Keep in mind that any (other) process may be the originating replica for an update. Some reflection shows that the full set of requirements is unattainable—a partition tolerant system simply cannot enforce both consistency and availability.

Theorem 3.2 (The CAP Theorem)**.** *In the presense of indefinite network partitions, a distributed system cannot guarantee both linearizability and eventual-availability.*

Proof. Technically, the proof is almost trivial. We give only the informal sketch here, leaving the interested reader to consult the more formal analysis by Gilbert and Lynch. The key technical assumption is that a processes' behavior can only be influenced by the messages it actually receives—it cannot be affected by messages that are sent to it but never delivered.

In Figure 12a, suppose the two processes are on opposite sides of a network partition, so that no information can be exchanged between them (even indirectly through a third party). If we just consider the execution of P_2 by itself, without P_1, linearizability would require it to read the value 2 for y. If we do consider P_1, linearizability requires that the read access to y must return 1. But if P_2 cannot send messages to P_1, then P_2's behavior cannot be influenced by the write access

to y, so it would still have to return 2, violating consistency. Alternatively, it could delay returning any result until it is able to exchange messages with P_1. But if the partition never recovers, P_1 will wait forever, violating availability. □

3.4.1 CAP for Sequential Consistency

Sequential consistency is a relaxation of atomic consistency, but not by much. The model is still too strict to enforce under partition conditions.

Lemma 3.3 (CAP for sequential consistency)**.** *An eventually-available system cannot provide sequential consistency in the presense of network partitions.*

Proof. The proof is an adaptation of Theorem 3.2. Suppose P_1 and P_2 form of CAP-available distributed system and consider the following execution: P_1 reads x, then assigns y the value 1. P_2 reads y, then assigns x the value 1. (Note that this is the sequence of requests shown in Figure 16a, but we make no assumptions about the values returned by the read requests). By availability, we know the requests will be handled (with responses sent back to clients) after a finite amount of time. Now suppose P_1 and P_2 are separated by a partition so they cannot read each other's writes during this process. For contradiction, suppose the execution is equivalent to a sequential order.

If $W(y, 1)$ precedes $R(y)$ in the sequential order, then $R(y)$ would be constrained to return to 1. But P_2 cannot pass information to P_1, so this is ruled out. To avoid this situation, suppose the sequential order places $R(y)$ before $W(y, 1)$, in which case $R(y)$ could correctly return the initial value of 0. However, by transitivity the $R(x)$ event would occur after $W(x, 1)$ event, so it would have to return 1. But there is no way to pass this information from P_1 to P_2. Thus, any attempt to consistently order the requests would require commuting $W(y, 1)$ with $R(x)$ or $W(x, 1)$ with $R(y)$, which would violate program order. □

3.4.2 Consequences of CAP

While the proof of the CAP theorem is simple, its interpretation is subtle and has been the subject of much discussion in the years since [1]. It is sometimes assumed that the CAP theorem claims that a distributed system can only offer two of the properties C, A, and P. In fact, the theorem constrains, but does not prohibit the existence of, applications that apply some relaxed amount of all three features. The CAP theorem only rules out their combination when all three are interpreted in a highly idealized sense.

In practice, applications can tolerate much weaker levels of consistency than linearizability. Furthermore, network partitions are usually not as dramatic as an indefinite communications blackout. Real conditions in our context are likely to be chaotic, featuring many smaller disruptions and delays and sometimes larger ones. Communications between different clients may be affected differently, with nearby agents generally likely to have better communication channels between them than agents that are far apart. Finally, CAP-availability is a suprisingly weak condition. Generally one cares about the actual time it takes to handle user requests, but the

CAP theorem exposes difficulties just ensuring the system handles requests at all. Altogether, the extremes of C, A, and P in the CAP theorem are not the appropriate conditions to apply to many, perhaps most, real-world applications.

The safety/liveness tradeoff The tension between consistency and availability is a prototypical example of a deeper tension in computing: that between safey and liveness properties [3, 6]. These terms can be understood as follows.

- **Safety** properties ensure that a system avoids doing something "bad' ' like violating a consistency invariant. Taken to the extreme, one way to ensure safety is to do nothing. For instance, we could enforce safety by never responding to read requests in order to avoid offering information that is inconsistent with that of other nodes.
- **Liveness** properties ensure that a system will eventually do something "good' ', like respond to a client. Taken to the extreme, one very lively behavior would be to immediately respond to user requests, without taking any steps to make sure this response is consistent with that of other nodes.

Note that "safety" is narrow sense, meaning a constraint on a system's allowable responses to clients, while liveness properties require the system to "do" something instead of delaying forever. Clearly, it is important for systems in safety-related applications to have some amount of liveness, and not just "safety" properties. Liveness and safety are both good.

Because of the tension between them, building applications that provide both safety and liveness features is challenging. The fundamental principle is that if we want to increase how quickly a system can respond to requests, eventually we must relax our constraints on what the system is allowed to return.

4 Desiderata and assumptions

4.1 Assumptions

- Only very loose clock synchronization. However, we may fairly assume that nodes can measure the passage of time. Real timestamps may be collected on data (for example, for later analysis), but the correct configuration of these clocks cannot be relied upon, particularly at lower levels of abstraction.
-

4.2 Desiderata

Having discussed some of the fundamental distributed systems issues that arise under real-world network conditions, we turn our attention to three desiderata we will use to frame and analyze the models discussed in Sections 6 and 7.

The CAP theorem, and others like it, place fundamental limitations on the consistency of real-world distributed systems. In the absense of a "perfect" system, engineers are forced to make tradeoffs. Ideally, these tradeoffs should be tuned for the specific application in mind—a protocol that works well in a datacenter might not work well in a heterogeneous geodistributed setting. This section lists three desirable features of distributed systems and frameworks for reasoning about or implementing them. We chose this set based on the particular details of civil aviation and disaster response, where safety is a high priority and usage/communication patterns may be unpredictable.

4.2.1 D1: Quantifiable bounds on inconsistency

A distributed application should quantify the amount of consistency it delivers. That is, it should (1) provide a mathematical way of measuring inconsistency between system nodes, and (2) bound this value while the system is available.

The CAP theorem implies that an available data replication application cannot bound inconsistency in all circumstances. When bounded inconsistency cannot be guaranteed, a system satisfying D1 may become unavailable. Alternatively, a reasonable behavior would be to continue providing some form of availability, but alert the user that due to network and system use conditions the requisite level of consisteny cannot be guaranteed by the application, leaving the user with the choice to assess the risk and continue using the system with a weaker safety guarantees.

4.2.2 D2: Accommodation of heterogeneous nodes

An application should not assume that there is a typical system node. Instead, the system should accomdate a diverse range of heterogeneous clients presenting different capabilities, tasks, and risk-factors.

One can expect a variety of hardware in the field. For example, wildfires often involve responses from many different fire departments, and it must be assumed that they are not always using identical systems. Different participants in the system may be solving different tasks, with different levels of access to the network, and they

present different risks. With these sorts of factors in mind, one should hope for frameworks that are as general as possible to accomodate a wide variety of clients.

4.2.3 D3: Optimization for a geodistributed wide area network

An application should be optimized for the sorts of communication patterns that occur in geodistributed wide area networks (WANs) under real-world conditions.

Consider two incidents. Wouldn't want to enforce needless global consistency, particularly if the agents in one area do not have the same consistency requirements for another area.

Network throughput has some (perhaps approximately linear) relationship with throughput. Communications patterns are likely far from uniform too. In fact, these two things likely coincide—it is often that nodes which are nearby have a stronger need to coordinate their actions than nodes which are far away. For example, consider manoeauvering airplanes to avoid crash.

5 Resilient Network Architectures

5.1 State of the art

- COWs and COLTs

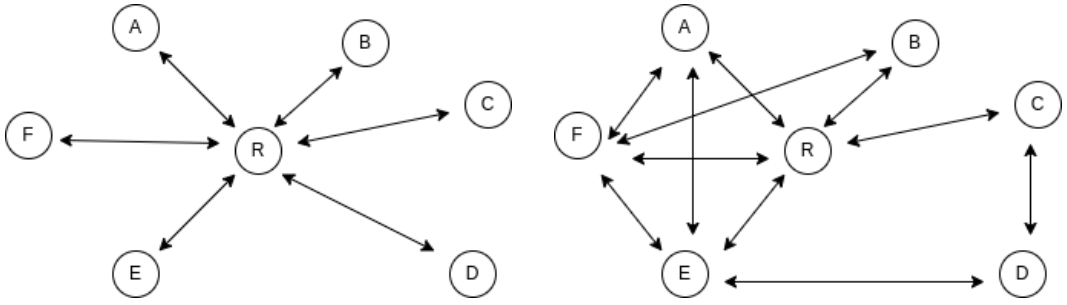
5.2 Ad-hoc networking

5.2.1 Physical communications

The details of the physical communication between processes is outside the scope of this memo. We make just a few high-level observations about the possibilities, as the details of the network layer are likely to have an impact on distributed applications, such as the shared memory abstraction we discuss below and in Section 6. For such applications, it may be important to optimize for the sorts of usage patterns encountered in real scenarios, which are affected by (among other things) the low-level details of the network.

The *celluar* model (Figure 17a) assumes nodes are within range of a powerful, centralized transmission station that performs routing functions. Message passing takes place by transmitting to the base station (labeled R), which routes the message to its destination. Such a model could be supported by the ad-hoc deployment of portable cellphone towers transported into the field, for instance.

The *ad-hoc* model (Figure 17b) assumes nodes communicate by passing messages directly to each other. This requires nodes to maintain information about things like routing and the approximate location of other nodes in the system, increasing complexity and introducing a possible source of inconsistency. However, it may be more workable given (i) the geographic mobility of agents in our scenarios (ii) difficult-to-access locations that prohibit setting up communication towers (iii) the inherent need for system flexibility during disaster scenarios.



(a) Cellular network topology (b) Ad-hoc network topology

Figure 17: Network topology models for geodistributed agents. Edges represent communication links (bidirectional for simplicity).

One can also imagine hybrid models, such as an ad-hoc arrangement of localized cells. In general, one expects more centralized topologies to be simpler for application developers to reason about, but to require more physical infrastructure and support. On the other hand, the ad-hoc model is more fault resistant, but more com-

plicated to implement and potentially offering fewer assurancess about performance. In either case, higher-level applications such as shared memory abstractions should be tuned for the networking environment. It would be even better if this tuning can take place dynamically, with applications reconfiguring manually or automatically to the particulars of the operating environment. This requires examining the relationship between the application and networking layers, rather than treating them as separate blackboxes.

5.3 Delay-tolerant networking

5.4 Ad-hoc DTNs

An interesting possibility is for the *network* to automatically configure itself to the quality-of-service needs of the application. For example, a client that receives a lot of requests may be marked as a preferred client and given higher-priority access to the network. If UAV vehicles can be used to route messages by acting as mobile transmission base stations, one can imagine selecting a flight pattern based on networking needs. For example, if the communication between two firefighting teams is obstructed by a geographical feature, a UAV could be dispatched to provide overhead communication support. Such an arrangement could greatly blur the line between the networking and application layers.

5.5 Software-defined networking

5.6 Verification of networking protocols

6 Continuous Consistency

Strong consistency is a discrete proposition: an application provides strong consistency or it does not. For many real-world applications, it evidently makes sense to work with data that is consistent up to some $\epsilon \in \mathbb{R}^{\geq 0}$. Thus, we shift from thinking about consistency as an all-or-nothing condition, towards consistency as a bound on inconsistency.

The definition of ϵ evidently requires a more or less application-specific notion of divergence between replicas of a shared data object. Take, say, an application for disseminating the most up-to-date visualization of the location of a fire front. It may be acceptable if this information appears 5 minutes out of date to a client, but unacceptable if it is 30 minutes out of date. That is, we could measure consistency with respect to *time*. One should expect the exact tolerance for ϵ will be depend very much on the client, among other things. For example, firefighters who are very close to a fire have a lower tolerance for stale information than a central client keeping only a birds-eye view of several fire fronts simultaneously.

Now suppose many disaster-response agencies coordinate with to update and propagate information about the availability of resources. A client may want to lookup the number of vehicles of a certain type that are available to be dispatched within a certain geographic range. We may stipulate that the value read by a client should always be 4 of the actual number, i.e. we could measure inconsistency with respect to some numerical value.

In the last example, the reader may wonder we should tolerate a client to read a value that is incorrect by 4, when clearly it is better to be incorrect by 0. Intuitively, the practical benefit of tolerating weaker values is to tolerate a greater level of imperfection in network communications. For example, suppose Alice and Bob are individually authorized to dispatch vehicles from a shared pool. In the event that they cannot share a message.

Or, would could ask that the the value is a conservative estimate, possibly lower but not higher than the actual amount. In these examples, we measure inconsistency in terms of a numerical value.

As a third example,

By varying ϵ, one can imagine consistency as a continuous spectrum. In light of the CAP theorem, we should likewise expect that applications with weaker consistency requirements (high ϵ) should provide higher availability, all other things being equal.

Yu and Vahdat explored the CAP tradeoff from this perspective in a series of papers [17–21] propose a theory of *conits*, a logical unit of data subject to their three metrics for measuring consistency. By controlling the threshold of acceptable inconsistency of each conit as a continuous quantity, applications can exercise precise control the tradeoff between consistency and performance, trading one for the other in a gradual fashion.

They built a prototype toolkit called TACT, which allows applications to specify precisely their desired levels of consistency for each conit. An interesting aspect of this work is that consistency can be tuned *dynamically*. This is desirable because one does not know a priori how much consistency or availability is acceptable.

The biggest question one must answer is the competing goals of generality and practicality. Generality means providing a general notion of measuring ϵ, while practicality means enforcing consistency in a way that can exploit weakened consistency requirements to offer better overall performance.

- The tradeoff of CAP is a continuous spectrum between linearizability and high-availability. More importantly, it can be tuned in real time.
- TACT captures neither CAP-consistency (i.e. neither atomic nor sequential consistency) nor CAP-availability (read and write requests may be delayed indefinitely if the system is unable to enforce consistency requirements because of network issues).

6.1 Causal consistency

Causal consistency is that each clients is consistent with a total order that contains the happened-before relation. It does not put a bound on divergence between replicas. Violations of causal consistency can present clients with deeply counterintuitive behavior.

- In a group messaing application, Alice posts a message and Bob replies. On Charlie's device, Bob's reply appears before Alice's original message.
- Alice sees a deposit for $100 made to her bank account and, because of this, decides to withdraw $50. When she refreshes the page, the deposit is gone and her account is overdrawn by 50. A little while later, she refreshes the page and the deposit reappears, but a penalty has been assessed for overdrawing her account.

In these scenarios, one agent takes an action *in response to* an event, but other processes observe these causally-related events taking place in the opposite order. In the first example, Charlie is able to observe a response to a message he does not see, which does not make sense to him. In the second example, Alice's observation at one instance causes her to take an action, but at a later point the cause for her actions appears to have occurred after her response to it. Both of these scenarios already violate atomic and sequential consistency because those models enforce a system-wide total order of events. Happily, they are also ruled out by causally consistent systems. The advantage of the causal consistency model is that it rules out this behavior without sacrificing system availability, as shown below.

Causal consistency enforces a global total order on events that are *causally related*. Here, causal relationships are estimated very conservatively: two events are potentially causally if there is some way that the outcome of one could have influenced another.

Lemma 6.1. *Sequential consistency implies causal consistency.*

Proof. This is immediate from the definitions. Sequential consistency requires all processes to observe the same total order of events, where this total order must respect program order. Causal consistency only requires processes to agree on events

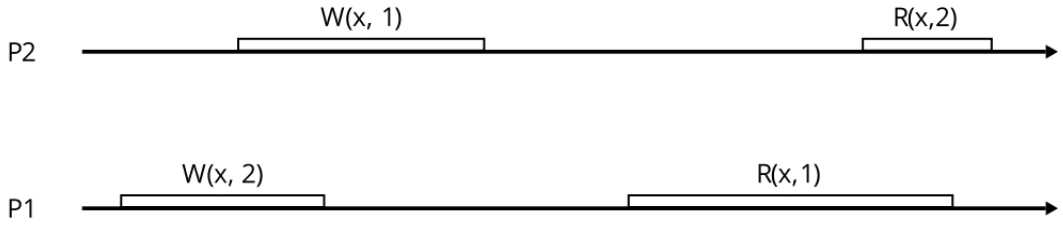


Figure 18: A causally consistent, non-sequentially-consistent execution

that are potentially causally related. Program order is a subset of causal order, so any sequential executions also respects causal order. □

However, causal consistency is not nearly as strong as sequential consistency, as processes do not need to agree on the order of events with no causal relation between them. This weakness is evident in the fact that the CAP theorem does not rule out highly available systems that maintain causal consistency even during network partitions.

Lemma 6.2. *A causally consistent system need not be unavailabile during partitions.*

Proof. Suppose P_1 and P_2 maintain replicas of a key-value store, as before, and suppose they are separated by a partition. The strategy is simple: each process immediately handles read requests by reading from its local replica, and handles write requests by applying the update to its local replica. It is easy to see this leads to causally consistent histories. Intuitively, the fact that no information flows between the processes also means the events of each process are not related by causality, so causality is not violated. □

Note that in this scenario, a client's requests are always routed to the same processor. If a client's requests can be routed to any node, causal consistency cannot be maintained without losing availability. One sometimes says that causal consistency is "sticky available" because clients must stick to the same processor during partitions.

The fact that causal consistency can be maintained during partitions suggests it is too weak. Indeed, there are no guarantees about the difference in values for x and y across the two replicas.

6.2 TACT system model

As in Section 3, we assume a distributed set of processes collaborate to maintain local replicas of a shared data object such as a database. Processes accept read and write requests from clients to update items, and they communicate with each other to ensure to ensure that all replicas remain consistent.

However, access to the data store is mediated by a middleware library, which sits between the local copy of the replica and the client. At a high level, TACT

will allow an operation to take place if it does not violate user-specific consistency bounds. If allowing an operation to proceed would violate consistency constraints, the operation blocks until TACT synchronizes with one or more other remote replicas. The operation remains blocked until TACT ensures that executing it would not violate consistency requirements.

$$\text{Consistency} = \langle \text{Numerical error, Order error, Staleness} \rangle.$$

Processes forward accesses to TACT, which handles commiting them to the store. TACT may not immediately process the request—instead it may need to coordinate with other processes to enforce consistency. When write requests are processed (i.e. when a response is sent to the originating client), they are only commited in a *tenative* state. Tentative writes eventually become fully committed at some point in the future, but when they are commited, they may be reordered. After fullying committing, writes are in a total order known to all processes.

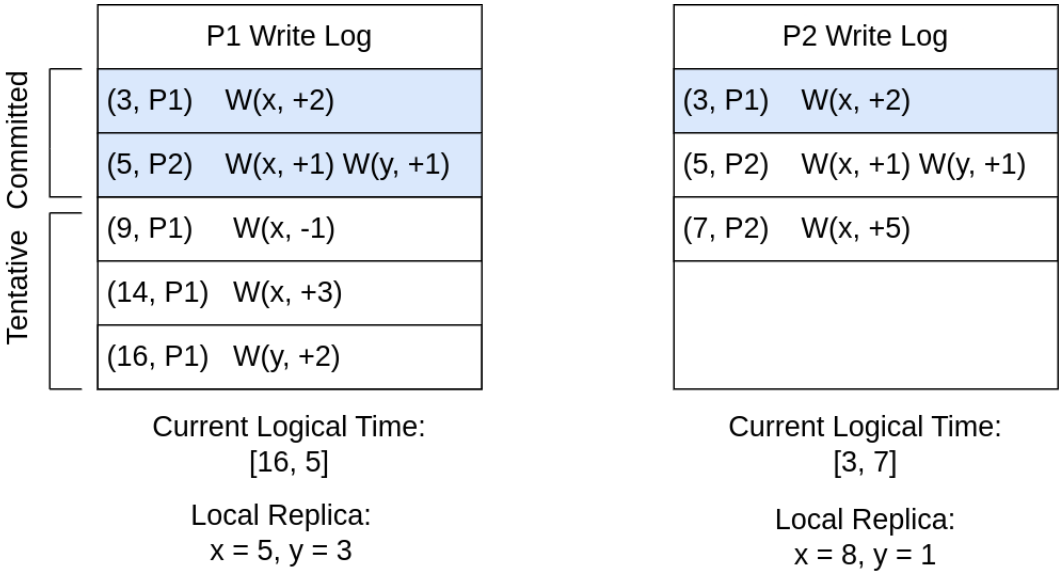


Figure 19: Snapshot of two local replicas using TACT

A write access W can separately quantify its *numerical weight* and *order weight* on conit F. Application programmers have multiple forms of control:

Consistency is enforced by the application by setting bounds on the consistency of read accesses. The TACT framework then enforces these consistency levels.

6.3 Measuring consistency on conits

Numerical consistency

Order consistency When the number of tentative (uncommitted) writes is high, TACT executes a write commitment algorithm. This is a *pull-based* approach which pulls information from other processes in order to advance P_i's vector clock, raising the watermark and hence allowing P_i to commit some of its writes.

Real time consistency

6.4 Enforcing inconsistency bounds

Numerical consistency We describe split-weight AE. Yu and Vahdat also describe two other schemes for bounding numerical error. One, compound AE, bounds absolute error trading space for communication overhead. In their simulations, they found minimal benefits to this tradeoff in general. It is possible that for specific applications the savings are worth it. They also consider a scheme, Relative NE, which bounds the relative error.

Order consistency

Real time consistency

6.5 Future work

7 Data Fusion

[16]

Strong consistency models provide the abstraction of an idealized global truth. In the case of conits, the numerical, commit-order, and real-time errors are measured with respect to an idealized global state of the database. This state may not exist on any one replica, but it is the state each replica would converge to if it were to see all remaining unseen updates.

We consider distributed applications that receive data from many different sources, such as from a sensor network (broadly defined). It will often be the case that some sources of data should be expected to agree with each other, but they may not. A typical scenario, we want to integrate these data into a larger model of some kind. Essentially take a poll, and attempt to synthesize a global picture that agrees as much as possible with the data reported from the sensor network.

Here, we need a consistency model to measure how successful our attempts are to synthesize a global image. And to tell us how much our sensors agree. Ideally, we could use this system to diagnose disagreements between sensors, identifying sensors that appear to be malfunctioning, or to detect abberations that necessitate a response.

7.1 Fusion centers

To be written.

7.2 Sheaf theory

7.2.1 Introduction to presheaves

Definition 7.1. A *partially order-indexed family of sets* is a family of sets indexed by a partially-ordered set, such that orders between the indices correspond to func-

tions between the sets.

We can also set $(P, \leq)$ *acts on* the set $\{S_i\}_{i \in I}$.

Definition 7.2. A *semiautomaton* is a monoid paired with a set.

This is also called a *monoid action* on the set.

Definition 7.3. A copresheaf is a *category acting on a family of sets*.

Definition 7.4. A presheaf is a *category acting covariantly on a family of sets*.

7.2.2 Introduction to sheaves

To be written.

7.2.3 The consistency radius

To be written.

8 Conclusion

To be written.

Bibliography

References

1. E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

2. California Department of Forestry and Fire Protection. Firefighter injuries and fatality: August 13, 2018, mendocino complex (ranch fire), 2023.

3. S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, sep 1985.

4. A. Fox and E. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

5. S. Gilbert and N. A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

6. S. Gilbert and N. A. Lynch. Perspectives on the cap theorem. *Computer*, 45(02):30–36, feb 2012.

7. A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2008.

8. S. MAGNUSON. 'coin of the realm': Military 'swimming in sensors and drowning in data'. *National Defense*, 94(674):36–38, 2010.

9. B. Mendelson. *Introduction to Topology: Third Edition*. Dover Books on Mathematics. Dover Publications, 2012.

10. National Interagency Incident Communications Division. Radio Cache. `https://web.archive.org/web/20231106025933/https://www.nifc.gov/about-us/what-is-nifc/radio-cache`. Accessed: 2023-11-10.

11. President's Council of Advisors on Science and Technology. Report to the president: Modernizing wildland firefighting to protect our firefighters, 2023.

12. R. Stickney. Cal fire youtube clip shows power of fire retardant drop. *NBC San Diego*, 2019.

13. M. Robinson. Sheaves are the canonical data structure for sensor integration. *Information Fusion*, 36:208–224, 2017.

14. M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., USA, 1994.

15. A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, Upper Saddle River, NJ, 2 edition, 2007.

16. L. Wald. Some terms of reference in data fusion. *Geoscience and Remote Sensing, IEEE Transactions on*, 37:1190 – 1193, 06 1999.

17. H. Yu and A. 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*, pages 75–84, 2000.

18. H. Yu and A. 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, USA, 2000. USENIX Association.

19. H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 123–133, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

20. H. Yu and A. 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*, pages 429–438. IEEE Computer Society, 2001.

21. H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, aug 2002.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01-12-2022	Technical Memorandum	

4. TITLE AND SUBTITLE
A Survey of Distributed Systems Challenges for Wildland Firefighting and Disaster Response

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Lawrence Dunn and Alwyn E. Goodloe

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, Virginia 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L–XXXXX

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2022–XXXXXX

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 64
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT

The System Wide Safety (SWS) program has been investigating how crewed and uncrewed aircraft can safely operate in shared airspace. Enforcing safety requirements for distributed agents requires 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. We survey topics ranging from consistency models and network architectures to data replication and data fusion, in each case focusing on the practical relevance of topics in the literature to the sorts of scenarios and challenges we expect from our use case.

15. SUBJECT TERMS
Distributed Systems, Formal Methods, Logic,

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Information Desk (help@sti.nasa.gov)
U	U	U	UU	46	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658