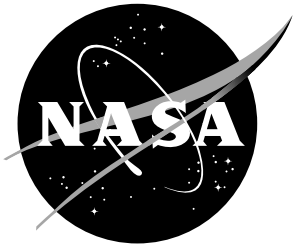
NASA/TM–2022–XXXXXX



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:

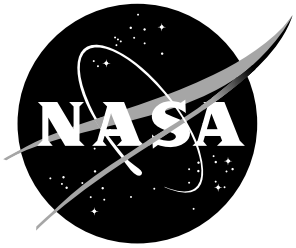
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference 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 [12]. 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 is flexible enough to 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–**??** 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 4–6 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 3.4 and 3.5, 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 5.

Section **??** 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 4 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 5 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") [25]. 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 6 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 [15]. 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 7 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 (particularly firefighting) scenarios that motivate the remainder 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 computer scientific problems raised when many distributed agents try to coordinate their actions over a widespread area.

The operational environment of wildfire suppression, hurricane relief, and other disaster settings 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 [12]. These shortfalls can be partly attributed to factors that are simply inherent to disaster response: remote locations, difficult terrain, damaged infrastructure, harsh weather, and limited battery power, to name a few.

Agents in the field generally experience high packet loss (when considering digital communication), garbled transmissions, and unpredictable latencies when passing information over the communication network(s). A conservative view suggests expecting the worst performance at particularly inopportune times, simply because the conditions that prompt urgent communication can be expected to correlate with those that make communication difficult. One possibility is that the disaster itself damages the communications infrastructure. Another example, which is true tautologically, is that a communications network is the most congested, and therefore the least available, precisely when everyone needs to use it. Both of these phenomena were famously exhibited in the immediate aftermath of the September 11^{th} terrorist attacks when sudden user demand and severed trunk cables brought New York public cell phone networks virtually to a halt [14]. Other networks along the East Coast, as well as dedicated networks for first responders, experienced similar effects.

From a systems perspective, an unreliable network presents a challenge for coordinating distributed agents. A root cause of this problem is that coordinated action requires some notion of consistency, i.e. agreement, among 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, which tasks should be prioritized, or which radios have been reserved from the NIICD radio cache [10] and for whom. The exact meaning of "consistency" varies between applications, but a general observation is that stronger standards for consistency 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 components that need to communicate may have to wait for network to deliver their messages, diminishing the efficacy and coherence of the system.

2.1 Communication and Safety

Many complications in the field are exacerbated by a poor communications environment. In this setting, agents must choose between long delays in sending and receiving information or acting with only limited knowledge. Typically they will experience some amount of both. Both options present safety challenges because operational safety, by nature, requires agents to gather information about their environment and react to it quickly and systematically. In other words, both inaction and uninformed action are problems. This turns out to be deeply related to a computer science phenomenon generally known as the safety/liveness tradeoff.

As a running example, we consider the use of firefighting airtankers, the largest of which are the Very Large Airtankers (VLATs), defined as those carrying more than 8,000 gallons of water or fire retardant [18]. The largest VLATs can carry more than 20,000 gallons, weighing about 170,000 pounds. This weight is typically dropped from a mere 250 feet above the tree canopy [18], though the complexity of the maneuver means errant drops are sometimes performed even lower than this, easily crushing a ground vehicle [13]. 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 running on their personal cell phones, or better yet something running on dedicated and more reliable equipment. A seemingly reasonable policy would be to prohibit VLATs 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: perhaps 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.

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:

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

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

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[2] of receiving a request.

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 actions are impeded to maintain $\mathbf{P}_{\text{safe}}$ at the cost of $\mathbf{P}_{\text{live}}$, allowing the fire to spread in the meantime.[3]

[2] In an interview with PBS, the Chief of Flight Operations for Cal Fire cited 20 minutes as the response time for aerial firefighting units within designated responsibility areas [20].

[3] A slight linguistic idiosyncrasy exhibited here is that liveness properties—not just "safety"

Besides the safety/liveness tradeoff, the previous example exhibits two other aspects of 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 feet of a drop zone. That is, it is not simply a matter of whether a danger is factually 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, 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 inflexible, 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 the information is only current to within 35 seconds. Clearly this is good enough information to authorize a drop, but strictly speaking the 5-second difference is a violation of $\mathbf{P}_{\text{safe}}$. When safety properties are this rigid, the system's behavior becomes overly sensitive to the particulars of the environment and therefore difficult to predict, which is precisely the kind of *discontinuity* that we aim to prevent. Our goal is to build reliable systems that perform well in a wide range of circumstances.

2.2 Communication Patterns in the Field

We now consider some of the communication patterns that occur in wildland firefighting. The layman reader may be surprised to learn that the state of the art is somewhat primitive, which is partly attributable to the fact that very little permanent infrastructure exists in this setting. This fact is also what makes wildfires an interesting and generalizable example for other kinds of civil disaster environments.

One trend we will draw attention to is a kind of "geospatial locality of reference" principle that system designers should take into account. By this, we mean the happy coincidence of two observations which, if not exactly guaranteed rules, are at least approximately true in many circumstances. The first observation is simple:

> $\mathbf{O}_1$: Agents with the most urgent need to coordinate their actions will tend to be located closer to each other and require the same kinds of information.

The second observation is as follows:

> $\mathbf{O}_2$: Agents that are located closer together will tend to have more reliable communications between them than agents that are far apart.

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.

> Conversely, information that must travel a long distance tends to be delayed or degrade in quality.

We will refer to the concomitance of these two facts as simply the "locality" principle. The reason the locality principle is crucial is that, as we see in Section 3, there are major theoretical and practical limits to how well *all* agents in the system can share *all* information with each other, i.e. how well a system can achieve global consistency. As luck would have it, in many cases this will not be required: it will be often be enough for *some* agents to share *some* information with each other, a fact that raises opportunities to optimize scare network resources. Of course, this does raise the question of how to decide which information must be shared with whom, and how to use this knowledge to best exploit the communication network. We will revisit this question, without the pretense of answering it conclusively, throughout Sections 4, 5 and 6. For now, we resume our examination of what communication patterns look like today.

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

As an alternative to using a radio, it is common for wildland firefighters in the field simply to shout commands and notifications to nearby personnel. This is a clear manifestation of the locality principle: a substantial amount of communication occurs directly between nearby firefighters working on the same or closely related tasks, and in some cases they are so nearby they can communicate without any network infrastructure at all. In a future environment where agents might be equipped with body-worn sensors and heads-up displays (HUDs) [4], this sort of local communication might be facilitated by simple low-power technologies such as Bluetooth [5], without the need for more sophisticated (and heavy) equipment.

Communication over a long distance requires infrastructural support, such as the use of cell towers and repeater stations. Typically, disaster response environments have scarce permanent infrastructure: in a wildland fire setting, perhaps a few repeaters mounted to a nearby watch tower. 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. An extremely 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. The difficulty of interoperability was another issue exhibited during the September 11^{th} attacks, which was a major impetus for the creation of the nationwide public safety broadband network (NPSBN) FirstNet [6]. However, we can only imagine that interoperability between different agencies and vendors will remain a challenge in the future.

[4] **CITATION NEEDED**
[5] **CITATION NEEDED**
[6] **CITATION NEEDED**

Figure 2: The Ironside Mountain lookout and radio repeater 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. **CITATION NEEDED**

Use of centralized infrastructure comes with the potential for widespread failure when the infrastructure breaks down. For example, in California, the Ironside Mountain lookout/repeater station, seen in Figure 2, was destroyed during the 2021 Monument Fire, which burned approximately 223,124 acres over 88 days [11]. 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[7], its loss prevented communication between operators on different sides of the ridge, in networking parlance creating a *partition* that lasted until crews could ascend the ridge to deploy a temporary station:

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

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

Vehicles on the ground Large numbers of ground vehicles are involved in wildfire suppression. A large wildfire response can involve up to 100 firetrucks distributed over a large geographical area. Bulldozers and similar vehicles are also commonly

[7] **CITATION NEEDED**

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. 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 low altitude. In other words, the demands are many and the margins for error are small. This sort of use case calls for more sophisticated coordination schemes between airborne and ground-based elements than solutions like ADS-B provide by themselves.

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

In some cases, planes from the Civil Air Patrol[8][9] 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 autonomously by base stations mounted to unmanned aerial vehicles (UAVs), which might perform additional functions such as tracking the fire perimeter. More generally, future systems should transparently facilitate exchanging information between agents in a decentralized fashion that is robust to the failure of any one component. In Section 4 we imagine a resilient ad-hoc digital network built from handheld and ground- and air-vehicle-mounted devices, permanent base stations, portable temporary infrastructure, and so on.

2.3 Towards the Future

So far we have said a lot about the state of disaster response today. A distributed system for this sort of challenge should be designed for the kinds of environments

[8] **CITATION NEEDED**

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

and conditions expected in the near- to medium-term future, so we briefly turn our attention to some of our expectations for this topic.

Perhaps the most prominent expectation for future disaster response events is a heavy reliance on *data*. Besides improvements to communications that facilitate information sharing, we expect advances in machine intelligence to greatly influence how this data is handled. Agents in disaster response environments will be both producers and consumers of data, and this data will need to processed by humans and machines in ways that agents can readily make sense of to support their decision-making. Our background research indicated many different kinds of data that could be valuable for responders. Just some of the kinds of information and communication we expect include the following:

- Free-form communication, especially recorded voice messages broadcast to many agents at once, which may need to be processed by machines to extract the most pertinent information into a more actionable format
- The exact or estimated location of victims, firefighters, vehicles, hazards, etc.
- Medical information gathered from victims, perhaps stored in and collected from digital triage tags[10]
- 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, rendezvous points, safety zones, and landing zones
- Availability and dispatching of assets, e.g. ambulances, airtankers, or crews on standby[11]

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. A superficially contradictory concern is that the information presented will be *overcomplete*, filled with petty details that distract agents from their important tasks.

In some ways, future systems for disaster response will bear resemblence to future systems for warfighting, such as the conceptual *Internet of Battle Things* (IoBT) [12]. Chiefly, agents "under extreme cognitive and physical stress" will be subject to a highly dynamic and dangerous environment. Various kinds of technology will assist humans by providing data to support sensemaking, but a contraindicating concern will be flooding agents with a "massive, complex, confusing, and potentially deceptive[13] ocean of information." To avoid "swimming in sensors and drowning in data" [8]:

> "Humans seek well-formed, reasonably-sized, essential information that is

[10] **CITATION NEEDED**

[11] NOTE Need to mention Monares et al 2011

[12] **CITATION NEEDED**

[13] While deliberately adversarial network behavior seems like less of a concern for disaster response agents than warfighters, we conjecture that a similar "fog of war" may lead to confusing or contradictory reports that share similarities with intentionally deceptive behavior.

> highly relevant to their cognitive needs, such as effective indications and warnings that pertain to their current situation and mission." [14]

The most distinctive feature of the Internet of Battle Things, which separates it from the everyday internet of things, is "the adversarial nature of the environment." To some extent this adversial behavior is common also to disaster response. We previously cited a real-world example of a critical communications station destroyed by wildfire, perhaps not unlike an attack by enemy forces.

Lest we overstate the similarities between a battlefield environment and civil disaster response, we expect that a distinctive feature of the latter is a greater emphasis on the preservation of scarce network resources. More so than a tactical military unit, a group of (say) volunteer firefighters has to make do with off-the-shelf equipment rather than purpose-built, best in class hardware like sophisticated satellite links. Dedicated logistical support, and even things like allocated radio frequencies, will likely be in shorter supply, while adverse conditions like inclement weather will be almost guaranteed. Therefore we expect a complex interaction between the high-level needs of distributed applications and low-level concerns about network resources. This is because only the applications have enough information to determine which data is the most important and must be shared with whom first, while only the network-level protocols have enough information and control to make prudent use of scarce network availability. In contravention to the common wisdom that applications should be relatively blind to network considerations—or conversely that network protocols ought to be transparent to applications—our setting calls for mechanisms allowing the two layers to have some influence over each other. This interaction between the network and applications will be considered in more detail in Sections 4, 5, and 6.

To give an example, 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*), or it may warn firefighters when they have strayed too far from an escape route or safety zone. Such information would be of high importance, and it would be worthwhile to expend network resources conveying this information to the relevant parties. It might also be nice for firefighters to have access to real-time information about the GPS location of every other firefighter. However, if this strains the network, then perhaps the exact location of teammates, but only the general location of other crews, is called for. If the network is extremely constrained, perhaps only information immediately relevant to preserving life should be sent in order to ensure the network is able to deliver this information quickly. The key point is that from where we stand, we cannot give a blanket rule determining which information is important, as this is partly a dynamic calculation influenced both by the criticality of the information to the task at hand and how much unused network capacity is available at that moment at that location.

[14] **CITATION NEEDED**

3 Introduction to Distributed Systems

In this section we review two core aspects of distributed systems theory, message passing and distributed shared memory, and address how these fit into the picture painted in Section 2. The primary reference we are following is the manuscript by Kshemkalyani and Singhal [7], particularly chapters 1, 3, 6, and 12. The main theme of the section is that distributed applications are sensitive to tradeoffs, typically between performance and a user-perceived notion of consistency.

In general, a distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved [7]. Our motivating examples are systems that facilitate cooperation between firefighters and other disaster response agents. The system components are typically computers, smartphones, sensors, or other kinds of communication equipment. These may be carried by individuals, mounted to ground or air-based vehicles, deployed as temporary or permanent infrastructure, or attached to autonomous devices. The kinds of tasks considered include navigating safely in close proximity, delivering resources to remote locations, suppressing fires, and collecting, processing, and disseminating data about the environment.

We consider two paradigms for framing distributed applications. In the message passing framework, the fundamental activity is sending and receiving information over the network. Alternatively, the fundamental activity of the shared memory framework is collaboratively updating data items shared by multiple agents. In either case, a chief concern is to shield users from the complex reality of the underlying system and "[appear] to the users of the system as a single coherent computer" [17], when it is really a distributed collection of loosely coupled devices. The main obstacle to this goal is that the network connecting the devices causes messages to be delayed unpredictably. When communication is delayed, information becomes out of date, and separated nodes will have different and possibly conflicting information. This situation tends to force a choice: delay operations and wait for messages to be passed, or proceed at the risk of diverging from total global consistency.

3.1 Message Passing

Singhal and Shivaratri [16] 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."

The most notable thing which makes a system "distributed" is that the nodes communicate by passing messages. Communication is facilitated by some form of network, a restriction implied by absence of a common memory. (Processes with common memory generally share information by writing it to a common memory location.) Section 2 explained why messages sent over the network experience a perceptible delay or *latency*, especially in the context of disaster response where

latencies may be on the order of minutes or even hours. First, passing messages requires the sender and receiver both to be within range of suitable access points to the network, which in these environments is likely true only intermittently. Then, messages must be sent over the air or through cables and processed, carried, and forwarded by any number of intermediate networking devices, and they must be retransmitted in case of unrecoverable errors during transit. In a tactical environment, packet loss can be on the order of %xx. [15]

The limitations of the network should be taken as environmental factors over which system designers can exercise only limited control. Consequently, network latencies must be tolerated to within the range of conditions experienced in the field. This raises fundamental questions of computer science: How, and how well, can agents coordinate their actions when communication between them is subject to unpredictable delays?

3.1.1 Assumptions about the Network

To focus on high-level questions about consistency and coordination, we introduce a few simplifying assumptions. We treat the communication network as a blackbox that sends messages from a single destination to a single receiver, subject to an unpredictable delay. We assume that if the same sender dispatches more than one message to a single destination, they arrive at their destination in the order they were sent. Below, this is called the FIFO (first-in, first-out) network model. Section 4 will consider the low-level details of how the network facilitates this kind of message passing model, but for now we take the fact for granted.

A major limitation of the FIFO model is that it does not constrain the order of delivery of messages from different senders or with different destinations. This means whenever group communication is considered between three or more agents, messages typically arrive in different orders at different destinations, at least at the lowest level of the network protocol stack used by each device. To facilitate a coherent abstraction for users and applications, higher-level protocols must be employed that account for out-of-order delivery and offer a more predictable interface to the network, such as by enforcing a common message order across an entire group. We shall just scratch the surface of this topic by discussing a basic mechanism common to many such protocols: using a system of "logical" clocks to track the causal order of events throughout the system.

3.1.2 FIFO and Causal Ordering

We model a system as a set $\mathcal{P} = \{P_i\}_{i \in I}$ of *processes* executing on independent, geographically dispersed computing devices, which we consolidate under the generic term *nodes*. It is important to note, in general, that different nodes can have different characteristics, such as capabilities and intended uses. In other words, the system is heterogeneous. For now we take a high-level view that treats nodes simply as blackboxes.

[15] **CITATION NEEDED**

Figure 3 shows several 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 depicts 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 starts off as a message *send event* m^{send} indicating the moment the message is sent over the network by its originating process; its delivery generates a receive event m^{recv}. The (send, receive) pairs are connected by an arrow and said to be corresponding events. Note that the subscripts on the messages only serve to disambiguate them to the reader. They are not part of the message and have no semantic importance.

The fact that arrows are diagonal, rather than vertical, represents the varying delay with which messages are delivered. Because of this delay, two messages m and m' may arrive in a different order than they were sent in. In Figure 3a, P_1 sends messages m_2 and m_4 respectively to P_3 and P_2, in that order, but the message to P_2 arrives first. This may depict a scenario where P_1 and P_3 are far apart and have a bad connection to each other. 3b depicts an extreme example where message m_1 is sent before all others but is last to be delivered. This diagram may depict a scenario where P_2 has an even better connection P_1, while P_3's connection has gotten worse, perhaps because the nodes are moving around the environment.

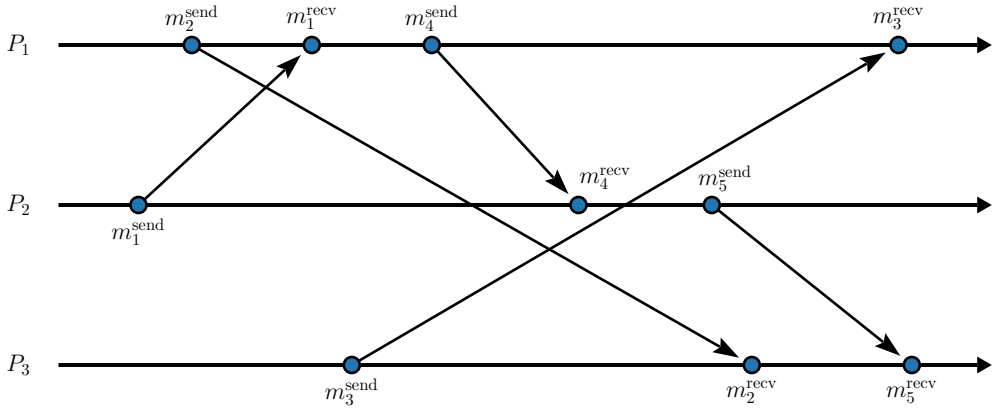
We assume just one exception to the fact that messages can arrive out of order. As highlighted earlier, at this level of abstraction it is reasonable to consider the FIFO (first-in, first-out) model, which means messages from the same sender to the same recipient arrive in the order they were sent in. FIFO ordering is one of the basic services that might be provided by the network transport protocol (see Section 4). However, this guarantee only applies to the relative order of two pairs of messages with the same source and destination; it does not constrain messages with different destinations or senders. The reader may wish to verify that all figures in this section satisfy FIFO: a violation of it would correspond to intersecting lines with the same (sender, receiver) pair.

Because FIFO does not apply to messages sent between three or more nodes, distributed applications can experience anomolies in message order. A distinct possibility is for messages to violate the *causal ordering* (CO) assumption. "Causality" in this context refers to the potential for information from one event to influence another. A key problem for distributed systems is that network latencies mean events with a causal relation may appear to happen in an illogical order, which makes tracking causality a key issue. We start by defining an order between events on the same process.

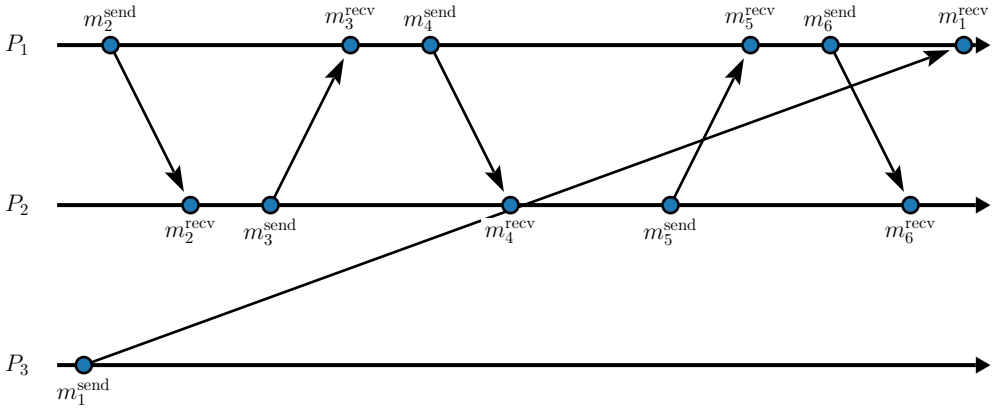
Definition 3.1. For two events e and e' that both occur in process P_i, we write $e <_{P_i} e'$ if e occurs before e' in P_i's worldline.

The previous definition is unambiguous because we assume events occur at discrete, non-overlapping moments at each process. Less clear is how to order events that occur on different processes. The most fundamental order considered is the *causal precedence* relation, also called Lamport's "happens before" relation [16].

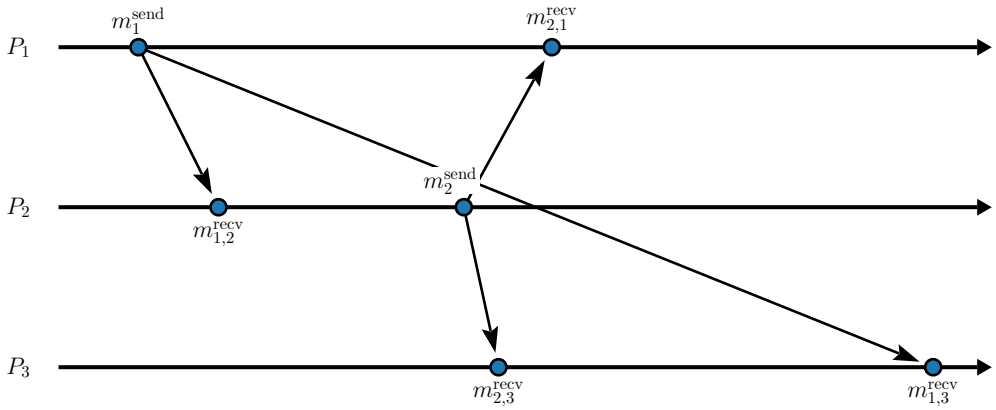
[16] **CITATION NEEDED**



(a) Depiction of latencies in message passing



(b) P_1 has a low-latency connection to P_2 and high-latency connection to P_3



(c) Broadcasting example satisfying FIFO but not CO

Figure 3: Message-passing time diagram examples

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

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

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

In words, there are three ways e could have causal precedence over e'. First it may be that the two events occur on the same process and e literally happens before e'. It may also be that e is a message send event and e' is the corresponding receive event at another process. Finally, it could be that e has precedence over some intermediate event e'' that has precedence over e', i.e. we consider the transitive closure of the previous two conditions. Informally, $e \to e'$ holds whenever one can put a finger on an event e in the time diagram and trace a path to e' by moving left-to-right along the worldlines or following the arrows connecting corresponding message event pairs.

The names "causal precedence" and "happens before" can be misnomers: $e \to e'$ can hold without it being the case that e really caused e' to happen, nor is it that case that if e literally occurs before e' that $e \to e'$ holds. Intuitively, $e \to e'$ conveys that information from e potentially affected or could have influenced event e'. A general desideratum is to avoid a scenario where $e \to e'$ holds in reality but, from a process's or user's perspective, it seems that e' actually occured before e. We define this formally before seeing an example.

Definition 3.3. (Causally consistent execution) A time diagram is *causally consistent*, or satisfies the causal ordering (CO) assumption, if the following property holds:

> For all messages m and n with the same destination, if $m^{\text{send}} \to n^{\text{send}}$, then $m^{\text{recv}} \to n^{\text{recv}}$.

In mathematical terms, the function mapping send events to corresponding receive events must be monotonic with respect to causal precedence.

For motivational purposes we consider a simple example. We slightly generalize the notion of a message and allow messages with multiple receipients (which could be realized by sending independent messages treated as one unit for present purposes). Our example is a group messaging application used by three firefighters, shown in Figure 3c, though in reality the processes may be machines sending messages at orders of magnitude faster than a human conversation. To post a message, a process sends it to the other two members of the group. For reasons explained above, the recipients generally do not receive this message at the same time.

Example 1. *In Figure 3c, we imagine the following conversation has taken place:*

P_1 *: "I need an ambulance at location A."*

P_2 *: "Understood, the last ambulance has been dispatched."*

However, P_3 witnesses P_2's answer before seeing P_1's question. This results in P_3 hearing a rather different conversation snippet:

P_2 : "Understood, the last ambulance has been dispatched."

P_1 : "I need an ambulance at location A."

From P_3's perspective, it seems that P_1 is requesting resources that are not available. The apparent conflict can lead to requests going unanswered, or possibly handled twice, leading to a chaotic situation and misallocation of resources. If these broadcasts represented machine-to-machine traffic between instances of a distributed application, it is likely the application would misbehave if not explicitly designed for this situation.

FIFO is vaccuously satisfied in Figure 3c because all four arrows have a distinct (sender, receiver) pair. However, the diagram violates Definition 3.3 because the send event of P_1's question has causal precedence over P_2's answer, while in P_3's worldline, the receive event of the answer has causal precedence over the receipt of question. It is possible to implement broadcast-ordering protocols that present the abstraction of multiple parties who receive group messages in the same order at all locations. Unsurprisingly, such protocols often require mechanisms for tracking the causal relation between events.

3.2 Timestamps and Synchronicity

It may appear that we can avoid the ambiguous situation described above by attaching to each event a *timestamp* tracking when the event occurred. Let us see why a naïve application of this idea fails. For each event e, let $C(e)$ denote the timestamp attached to that event. The fundamental property we want to satisfy is the following one:

Definition 3.4. A system of timestamps satisfies the *clock consistency condition* if the following monotonicity property holds:

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

In words, CC requires that if e causally precedes e', then e should have a lesser timestamp. In Example 1, we had two send events $m_1^{\text{send}} \to m_2^{\text{send}}$, so the clock consistency condition would require that

$$C(m_1^{\text{send}}) < C(m_2^{\text{send}}).$$

Now message m_1 is timestamped and sent by P_1, while m_2 is timestamped and sent by P_2. A hallmark of distributed systems is we cannot assume processes have instantaneous access to a common physical clock, which means the inequality cannot be guaranteed unless P_1 and P_2 happen to have two different clocks that are synchronized. Whether we can assume clocks are synchronized depends on the exact environment and the level of synchronicity required, but generally sychronization is either unavailable or requires additional mechanisms to prevent clocks from falling out of sync.

Physical clocks, especially inexpensive ones in consumer-grade devices, suffer from *drift*, which is to say they do not all run at the same rate. They are also notoriously prone to misconfiguration, say if a device administrator sets the date, time, timezone, or daylight saving time policy (etc.) incorrectly. The devices we are considering may also spend a long time unpowered in storage, during which time they may not maintain an always-on clock. For all these reasons, ordinary device clocks by themselves are not very reliable. They can be made more reliable using protocols like the venerable Network Time Protocol (NTP) [17]to bring them into synchronization with respect to authoritative atomic clocks. Over a standard internet connection, the NTP works well enough to bring a device clock into synchronization with atomic clocks within 100ms, typically better. However, though NTP is robust to the network environment, this may not extend into the sorts of highly intermittent networks considered here and in Section 4. We will consider further issues of synchronization in that section.

For some coarse-grained purposes, possibly including the database replication framework in Section 5, it may be enough for devices to maintain clocks that are only somewhat precisely synchronized. For other purposes, such as enforcing a total order on a sequence of broadcasts among large distributed groups, sufficiently precise synchronization of clocks is an unrealistic assumption.

It also happens that merely timestamping messages, even with perfectly sychronized clocks, is not 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:

Definition 3.5. A system of timestamps satisfies the strong clock consistency condition if the following property holds:

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

SC strengthens CC by stipulating that we can always compare two timestamps to infer whether the events are causally related. This condition cannot be achieved using simple timestamps even from synchronized clocks. This is because physical timestamps always assign a relative order between any two messages. However, we have seen that some events are physically ordered but not causally related, and simply comparing physical timestamps does not reveal the lack of causal relation. An example of this situation is found in Figure 3b, where m_1^{send} physically precedes all other events but does not causally precede any event except m_1^{recv}.

3.3 Logical Clocks

Distributed applications can systematically track causality by employing a system of *logical clocks*, which usually measure the logical flow of time by storing non-negative integers that are advanced according to certain rules. Three major variants are common: scalar, vector, and matrix clocks. These fall on a kind of spectrum, in that scalar clocks are simple but provide the most coarse-grained information, while

[17]**CITATION NEEDED**

vector and matrix clocks provide increasingly more information but impose greater overheads.

3.3.1 Scalar clocks

Scalar clocks, introduced by Lamport[18], require each process P_i to maintain a single scalar value C_i, a non-negative integer initialized at 0. There are two update rules:

1. Before 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 send event. The value is sent ("piggybacked") as part of the message metadata for use by the receiver.

2. When P_i receives a message timestamped with value C, the process updates C_i according to the rule

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

   The updated value is the timestamp associated with the receive event.

Figure 4 depicts the same events in Figure 3 alongside the scalar timestamp that would be assigned to each event. It is not hard to prove, and the reader should verify, that scalar clocks satisfy the clock consistency condition (CC). Intuitively, this is because a node ensures its clock is incremented before every event and always greater than the timestamp piggybacked on top of any message it receives.

Lemma 3.1. *A system of scalar clocks satisfies* $e \to e' \implies C(e) < C(e')$.

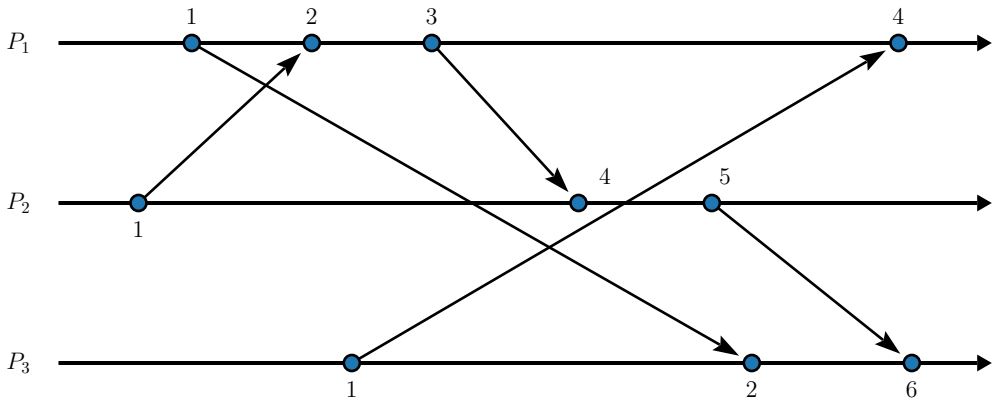
However, scalar clocks do not satisfy the strong condition (SC). That is, even if the timestamp of one event e' is greater than that of e, the possiblity is left open that e does not have causal precedence over e'. For example, examining Figures 3b and 4b, $C(m_1^{\text{send}})$ has a globally minimal timestamp value of 1, which is less than the timestamp of every other event except m_2^{send}. Nonetheless, the sending of m_1 is causally unrelated to every event but its own delivery.

Returning to the previous example, Figure 3c and 4c, because m_1^{send} causally precedes m_2^{send}, we find

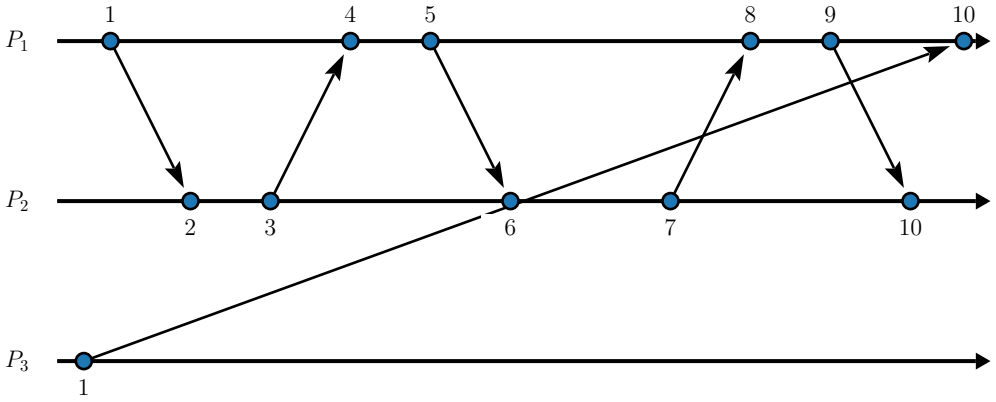
$$C(m_1^{\text{send}}) = 1 < 5 = C(m_2^{\text{send}}).$$

Using this fact P_3 can learn (by examining piggybacked timestamps) that P_2's response could not causally precede P_1's question, partially resolving the ambiguity posed earlier. However, the scalar clock regimen does not given P_3 enough information to conclusively affirm that the question causally precedes the answer: the opposite is ruled out, but it is possible the events are causally independent.

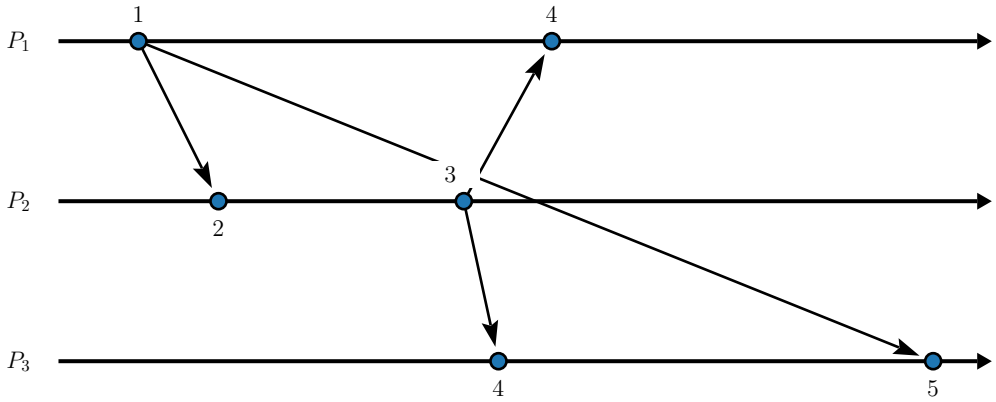
[18] **CITATION NEEDED**



(a) Scalar clocks for 3a



(b) Scalar clocks for 3b



(c) Scalar clocks for 3c

Figure 4: Scalar clock examples

Scalar clocks achieve the clock consistency condition without synchronized physical clocks. They are often used when an application needs to establish a shared global ordering among all events in the system, such as in state machine replication (SMR) protocols. Because timestamps are not globally unique, they do not carry quite enough information for recipients to compute a shared total order by themselves. A simple way to obtain a total order is to use an arbitrary tie-breaking mechanism for deciding how to order two (necessarily causally unrelated) events that happen to share a common scalar clock timestamp, typically by comparing the (necessarily distinct) identifiers of the processes where they took place. Indeed, this application was the motivating example when scalar clocks were introduced by Lamport.

However, like physical timestamps from synchronized clocks, scalar clocks are insufficient for applications that need to perform detailed causality tracking that relies on the stronger clock consistency condition.

3.3.2 Vector clocks

Vector clocks require each process to store one scalar value for each process in the system. That is, for 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 $\mathrm{vt}_i[x] = 0$. As with scalar clocks there are two update rules:

1. Before a new 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 piggybacked as part of the message's metadata so the receiver can use it.

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

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

   After this, 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 receive event.

According to these rules, the i^{th} component $\mathrm{vt}_i[i]$ of P_i's vector clock acts somewhat like a scalar clock: we call it the *local time* for P_i. A slight difference between the local time and a scalar clock is that after receiving a message, P_i's local time is not necessarily greater than the sender's local time that was piggybacked with the message. (What matters is that P_i's overall vector time is greater.) For all other $j \neq i$, the j^{th} component $\mathrm{vt}_i[j]$ of P_i's clock represents P_i's *estimate* of P_j's local time. This estimate is always a lower bound, since P_j's actual local time may have advanced in the meantime.

Figure 5 redepicts the diagrams in Figure 3 showing the vector timestamp that would be assigned to each event.

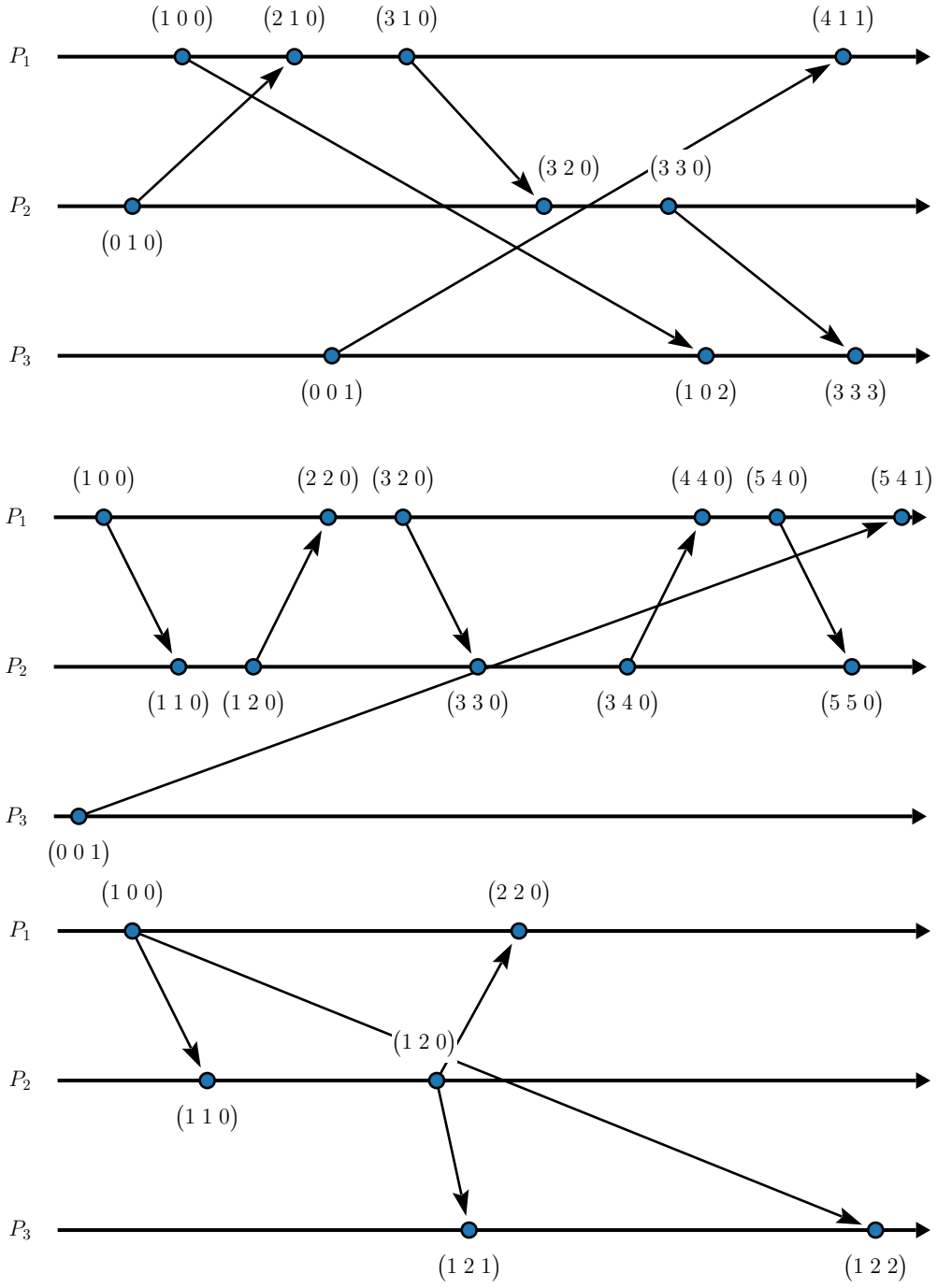


Figure 5: Vector clock examples

Vector clocks are compared component-wise. Crucially, this only gives a partial order among timestamps, as one vector may be greater than another in some components and less in others.

Definition 3.6 (Vector comparison)**.** Let v, w be two vector clocks. We define the following relations:

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

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

The reason to consider vector clocks over scalar clocks is precisely that they satisfy SC, so they can be used to unambiguously decide whether two timestamped events had causal influence over the other.

Lemma 3.2. *Vector clocks satisfy the strong clock consistency condition. That is, where $C(e)$ is the vector timestamp of an event, then*

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

For reasons of space we omit a proof of the precending lemma. The direction $e \to e' \implies C(e) \prec C(e')$ is straightforward. More tedious is the right-to-left direction $C(e) \prec C(e') \implies e \to e'$, but the key idea is that if $C(e) \prec C(e')$, then it is possible to follow a backwards chain of send and receive events connecting e to e', hence $e \to e'$.

3.3.3 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. The justification for matrix clocks is somewhat subtle, but we shall see a clear example in Section 5 of why tracking this extra information can prove useful.

Matrix clocks subsume vector clocks, so they can be used whenever the strong consistency condition is needed. The reader may then be left wondering what extra utility is provided. 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. If logical clocks are roughly thought of as a sense of "time," the matrix clocks are useful when processes need to know what time other processes think it is. More concretely, when (CONDITION), the process knows (STATE). In Section 5 we will see an example where this sort of lower bound estimate of other process's clocks is important for distributed database replication.

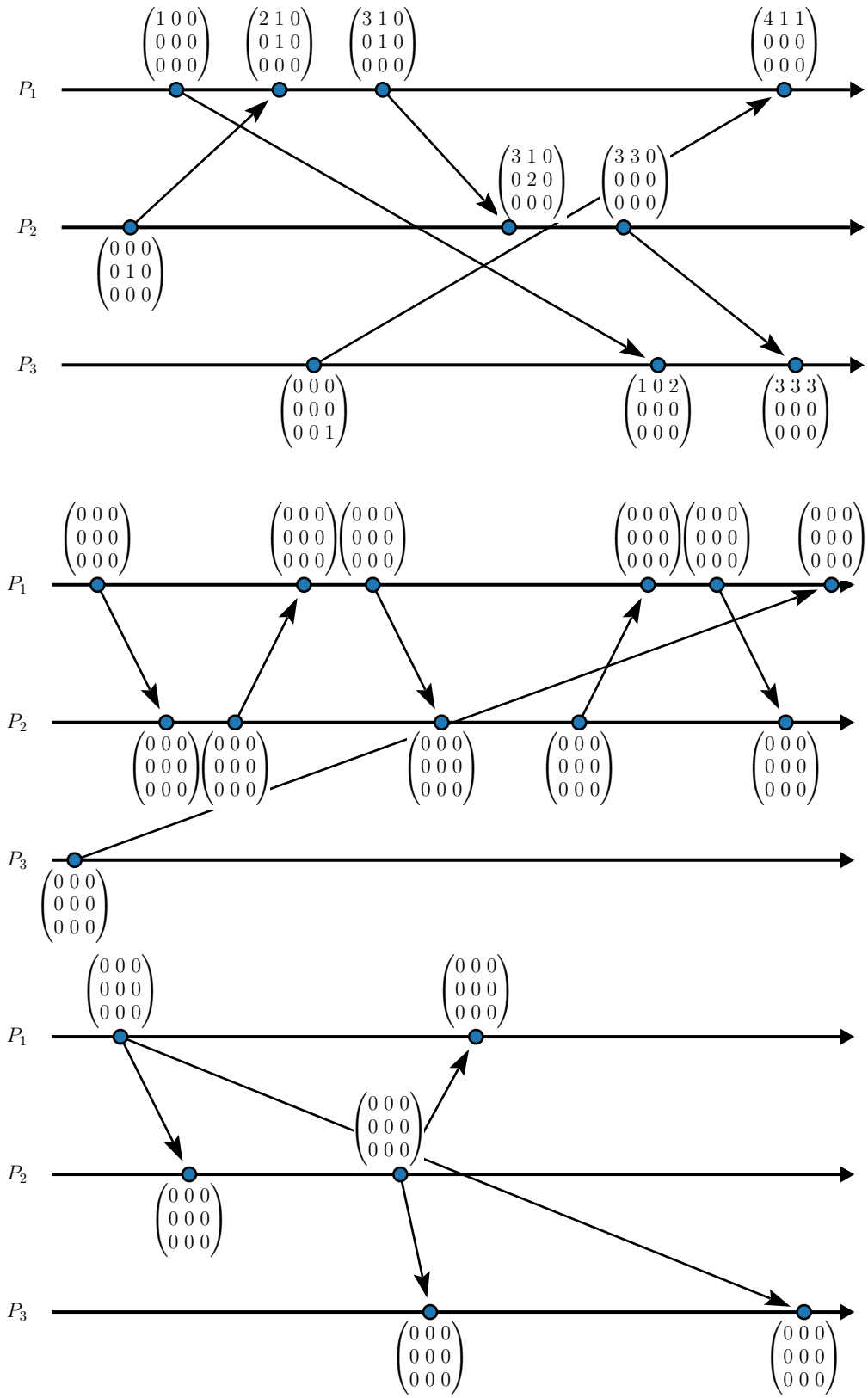


Figure 6: Matrix clock examples

3.4 Shared Memory

Programming distributed applications in terms of sending and receiving messages over the network is a complex task. As we have seen, it raises issues of synchronization and consistency that must be dealt with in order to provide understandable system behavior. In this section we summarize a simpler but in some sense equivalent paradigm called the *distributed shared memory* (DSM) abstraction.

In the DSM model, programmers think in terms of reading and writing values to memory locations (one might also say *variables*) rather than directly transmitting messages over the network. The key feature is that all instances of the application act as if they are reading and writing from the same memory locations, when in reality they are running on distributed computers that do not share memory. Transparently and behind the scenes, a "middleware" memory management protocol layer translates the read/write requests into send/receive requests that facilitate the abstraction of a common virtual memory space. The programmer does not have to know how this is achieved exactly, but it is critical for them to know what guarantees the memory management layer provides. As is typically the case, systems can differ in this respect because they make different tradeoffs between safety guarantees and performance.

3.4.1 Shared memory time diagrams

Figure 7a shows a time diagram for the shared memory abstraction, not unlike the diagrams in Figure 3 for message passing. The diagram depicts a series of operations, which come in two types:

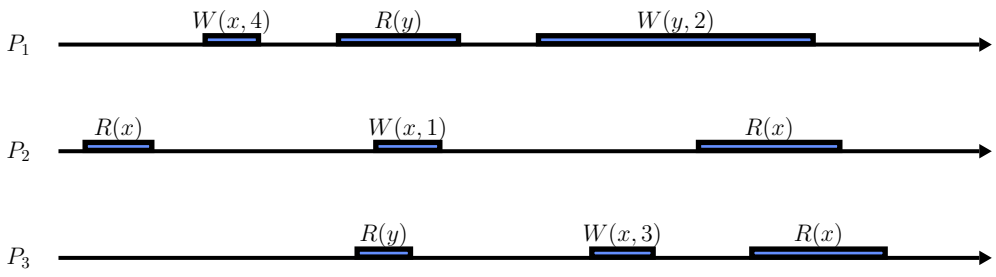
Reads A *read* request $R(x)$ indicates reading the value in memory at location x, which returns some value v. When it is important to indicate the return value, we sometimes notate read requests as $R(x, v)$.

Writes A *write* operation $W(x, v)$ indicates writing value v to memory location x. In pseudocode the notation might be $x := v$ or $x \leftarrow v$.

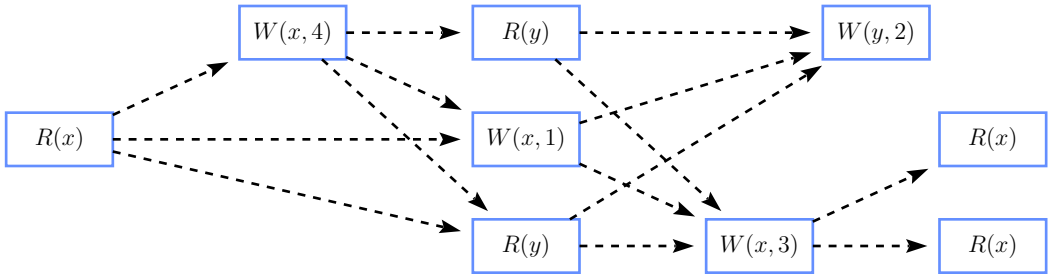
Each request consists of a horizontal span, shown as a shallow box, along the worldline of a process. The span represents the duration when the request is executing, beginning when the request is submitted by the application and ending when the operation has finished. From the programmer's perspective, if the read/write primitives are blocking operations, the execution is when the memory management layer is running and the main application is blocked. During this time, the memory management layer might coordinate with other processes in the background over the network. During a read request $R(x)$ it may, for example, send messages to another server to lookup the "most current" value of location x. Such behind-the-scenes coordination is transparent to the programmer and not depicted in the time diagrams.

3.4.2 Semantics and consistency

For a non-distributed application, i.e. one running on a single computer, it is clear how read and write requests should be interpreted. Mainly, a read request $R(x)$ should return the most recent value v that was written to x by a write $W(x, v)$,



(a) Time diagram for memory operations



(b) The directed acyclic graph induced by external order.

Figure 7: Shared memory time diagram

or resort to some default behavior if no such write exists. This interpretation is unambiguous because we assume that operations running on a single process do not overlap in time, so it always makes sense to ask which of two events happened before the other.

Example 2. *Consider Figure CITE, depicting just a single process. Since there is no ambiguity in the order of events, it is clear that this process should execute as such (note that we indicate the values returned by each read request):*

$$W(x,0), W(y,5), R(x,0), W(x,3), R(y,5), R(x,3).$$

In the distributed context, the metaphorical wrench in the works is that operations by different processes can and frequently will overlap, so by default there is no total order that can be used to compare events. Without such a comparison, the notion of "most recent" operation is ambiguous, which makes it difficult to say exactly how the memory requests should even be interpreted.

Examine Figure CITE. Because the two writes overlap in physical time, we cannot say that either $W(x,3)$ or $W(x,5)$ happened first, so it is unclear whether the read request should return 3 or 5. However, it seems intuitively clear that the programmer should be confident that $R(x)$ will return 3 or 5, and not, say, 37. But consider the second request $R(x)$ running on process 2. Can the programmer be confident that both requests will return the same value, or might they seemingly "disagree" with each other? Finally, consider the second request $R(x)$. Is it possible that this request reads a different value than the read immediately preceding it? Ultimately, it is possible to consider different ways of answering these questions. The role of different *memory models* is to constrain exactly which possibilities are allowed and which are prohibited, and so the same program can execute differently depending on what memory model is offered by its execution environment.

As before, it is tempting to imagine assigning a physical timestamp to reads and writes to establish an order among events, resolving the ambiguities above. However, we have seen that physical clocks are usually not so precisely sychronized as to allow meaningfully comparing timestamps from different nodes, so this is not a total solution.

It also happens that, for performance reasons, memory models commonly allow distributed operations to have semantics that do not strictly conform to the physical order of events. Indeed, below we show how sequential consistency (Definition 3.13), one of two "strong" models, allows for this possibility. Yet even this model imposes too many constraints to be workable for networking environments as unpredictable as ours.

3.4.3 Concurrency and External order

Definition 3.7. 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.

One fundamental relation in the DSM setting is *external order*. Intuitively, it is the partial order that orders non-overlapping events by their physical times, but does not assign an ordering to events whose executions overlap in physical time.

Definition 3.8. Recall that an irreflexive partial order is a binary relation $<$ such that $A \not< A$, $A < B \implies B \not< A$, and $A < B, B < C \implies A < C$, where $x \not< y$ means it is not the case that $x < y$.

Definition 3.9. 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*.

Figure 7b shows the directed acyclic graph (DAG) induced by external order on the operations of 7a.

We assume processes handle operations one-at-a-time, so 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*.

3.5 Memory Consistency Models

Since the goal is for the system to present the illusion of a reading and writing from a common memory, the observed result should be equivalent to some interleaving of the requests into a linear order. If each processes P_i of N total processes issues r_i requests for some non-negative integer r_i, observe there are a total of

$$\frac{\left(\sum_{i=1}^{N} r_i\right)!}{\prod_{i=1}^{N} r_i!}$$

such interleavings. The question we must consider is which of these are allowed.

3.5.1 Linearizability

Linearizability, the strongest consistency model, can be concisely defined as providing the appearance that "each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response." [19]The same condition is known variously as atomic consistency, strict consistency, and sometimes external consistency. In the context of database transactions (which come with other database-specific guarantees, like isolation), the analogous condition is called strict serializability.

A linearizable execution is defined by three features.

[19]**CITATION NEEDED**

Definition 3.10 (Linearizable execution)**.** A *linearizable execution* is one satisfying the following three conditions.

1. All processes act like they agree on a single, global total order defined across all accesses.
2. This sequential order is consistent with the actual external order.
3. 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.

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

Definition 3.11. 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.

Definition 3.12 (Linearizable system)**.** A shared-memory application is *linearizable* if all possible executions of that system are linearizable executions.

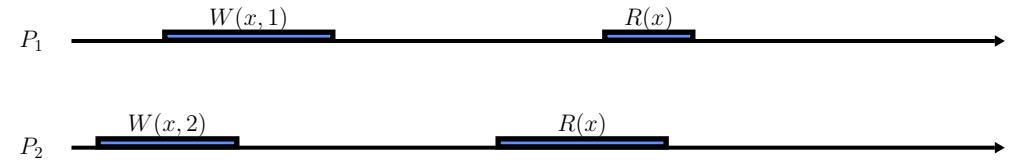
Figure 13a shows a prototypical example of a linearizable execution. We assume that all memory locations are initialized to 0 at the system start time. Intuitively, it should appear to an external observer that each access instantaneously took effect at some point between its start and end time. Hence, 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 13b shows an non-linearizable execution that returns stale data instead of reflecting the write access to y on $P2$.

Linearization points are demonstrated in Figure 14. 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 15).

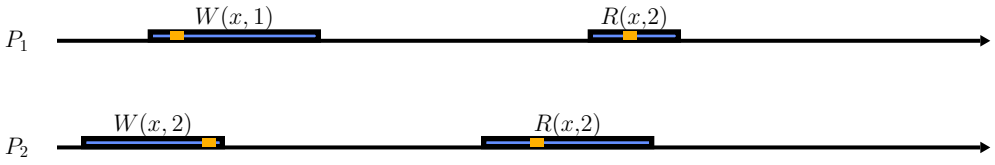
3.5.2 Sequential consistency

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.

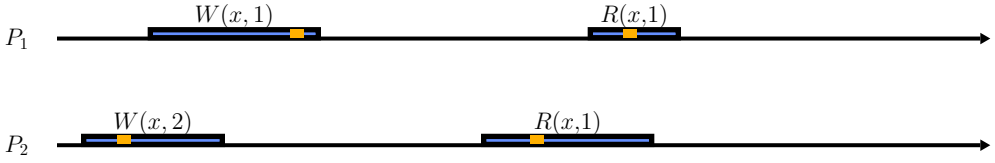
A sequentially consistent system ensures that any execution is equivalent to some global serial execution, even if this is serial order is not the one suggested by the real-time ordering of events. When real-time constraints are not important,



(a) Linearizable execution



(b) One possible linearization



(c) Another possible linearization

Figure 8: Linearizabile executions

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 3.13. 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.

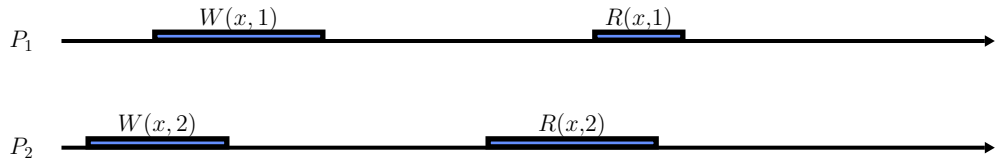
Visually, sequential consistency allows reordering an execution by sliding events along each process' time axis like beads along a string. Two events from the same process cannot pass over each other as this would violate program order, but events on different processes may be commuted past each other, 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.

It may seem strange to consider executions such as the one shown in REF in which operations appear to take effect at different times for different processes, or at times that do not agree with external order. The reader should remember that in the background, these processes would be engaged in message-passing over the network and are therefore subject to all the complexities previous discussed in Section **??**, including delayed and out-of-order messages. Looser requirements by the memory model impose fewer constraints on the message passing layer requiring less coordination, and allowing for greater performance.

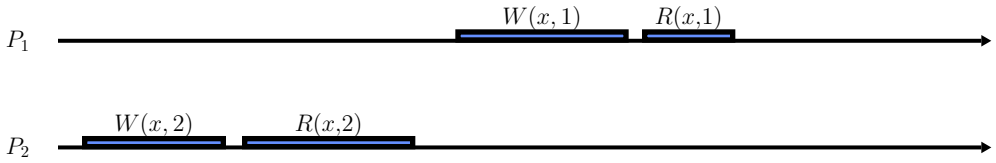
Lemma 3.3. *A linearizable execution is sequentially consistent.*

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

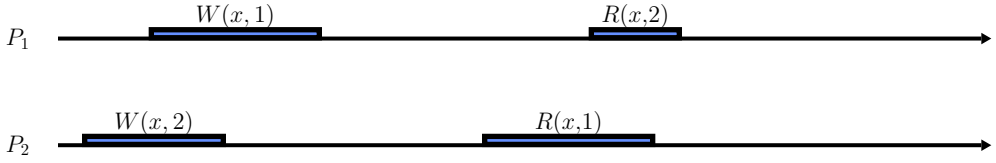
The converse of Lemma 3.3 does not hold. For example, Figure 16a was previously shown (Figure 15a) as a nonlinearizable execution. However, it is sequentially consistent, as evidenced by the interleaving in Figure 16b that slides the events $W(x, 1)$ and $R(x, 2)$ past each other.



(a) Sequential, non-linearizable execution



(b) Demonstration of sequential consistency of the above execution



(c) Nonsequential execution

Figure 9: Sequential and non-sequential executions

3.5.3 Causal consistency

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

3.6 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 3.4 (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 13a, 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. □

CAP also holds for sequential consistency.

Lemma 3.5 (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 3.4. 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 17a, 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. □

3.6.1 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.

3.7 Summary

We have seen that a distributed system is built from geographically distant components that must communicate over a network. Sending messages over the network causes messages to suffer unpredictable delays. Particularly in the context of broadcasts sent to multiple members of a group at once, this varying delay often causes messages to arrive in different orders to different members of the group, which can lead to chaos if a message-ordering discipline is not imposed.

As part of taming this chaos, it is critical for distributed systems to track the causal precedence relation between events. Physical clocks cannot usually be relied upon to maintain adequate synchronization for this purpose. Instead, logical clocks may be employed. The different paradigms—scalar, vector, and matrix clocks—vary in how much information they track and how much overhead they impose on the system with respect to bandwidth and storage space. The latter two require nodes to know about every other member of the group, as is typical for many mechanisms in distributed systems. If groups can change dynamically, as in our scenarios, then a group membership protocol is also needed.

Programmers may find it easier to frame distributed applications in terms of reading and writing from a shared pool of virtual memory instead of sending messages over a network. However, the fact that many nodes can access this virtual memory at the same time leads to non-determinism in how the memory accesses are ordered, which makes it non-trivial to decide what it means for the system to be consistent. The two strongest notions of consistency—linearizability and sequential consistency—essentially provide the illusion of a single source of truth, but the CAP theorem makes it virtually impossible to realize these consistency models in the kinds of chaotic networks we are considering. The causal consistency model ensures a minimum of coherence and has the advantage that it is not subject to the limitations of the CAP theorem. However, it makes no guarantees about how far apart replicas of the same data may diverge, which makes this model too weak for the kinds of safety-related applications we have in mind.

4 Resilient Network Architectures

A priori, a network may not deliver a message at all. Alternatively it might deliver the message multiple times, for example if a device is unaware that a message has already been delivered. In either case, the network itself does not alert the sender or receiver to the fact—the responsibility for detecting such conditions belongs to the sorts of protocols considered here and in Section 4.

If we were to inspect the network blackbox, we would expect to find so-called

"transport" protocols like TCP[20]being used to provide the abstraction of reliable delivery over an otherwise unreliable network. For example, they might add meta-data to messages that allow the receiver to arrange them in their intended order, discard duplicates, and verify their integrity. Transport protocols might also handle retranmission when messages become corrupted in transit. We consider these kinds of low-level details in Section 4; for now, one could say we are working at a level of abstraction above the transport layer. Note that transport reliability mechanisms contribute to the latency in passing messages, so they cannot solve the problems under consideration in this section.

4.1 Ad-hoc networking

4.1.1 Physical communications

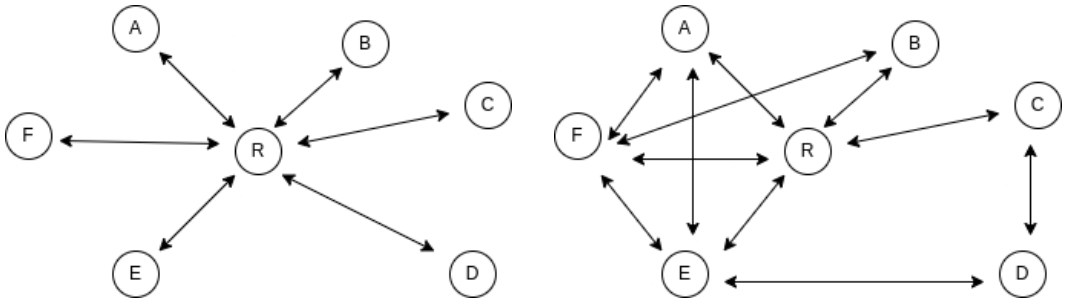
The details of the physical communication between processes is outside the scope of this memo. We make just a few high-level observations about the possibilities, as the details of the network layer are likely to have an impact on distributed applications, such as the shared memory abstraction we discuss below and in Section 5. 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 10a) 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 10b) 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.

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 complicated 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.

[20]**CITATION NEEDED**



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

Figure 10: Network topology models for geodistributed agents. Edges represent communication links (bidirectional for simplicity).

4.2 Delay-tolerant networking

4.3 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.

4.4 Software-defined networking

4.5 Verification of networking protocols

5 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 [21–25] 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).

5.1 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 5.1. *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 that are potentially causally related. Program order is a subset of causal order, so any sequential executions also respects causal order. □

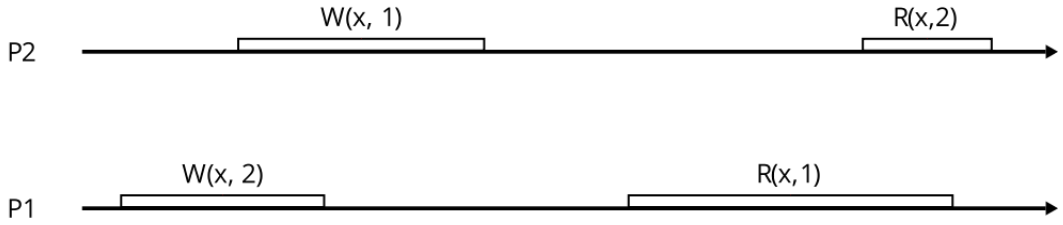


Figure 11: A causally consistent, non-sequentially-consistent execution

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 5.2. *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.

5.2 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.

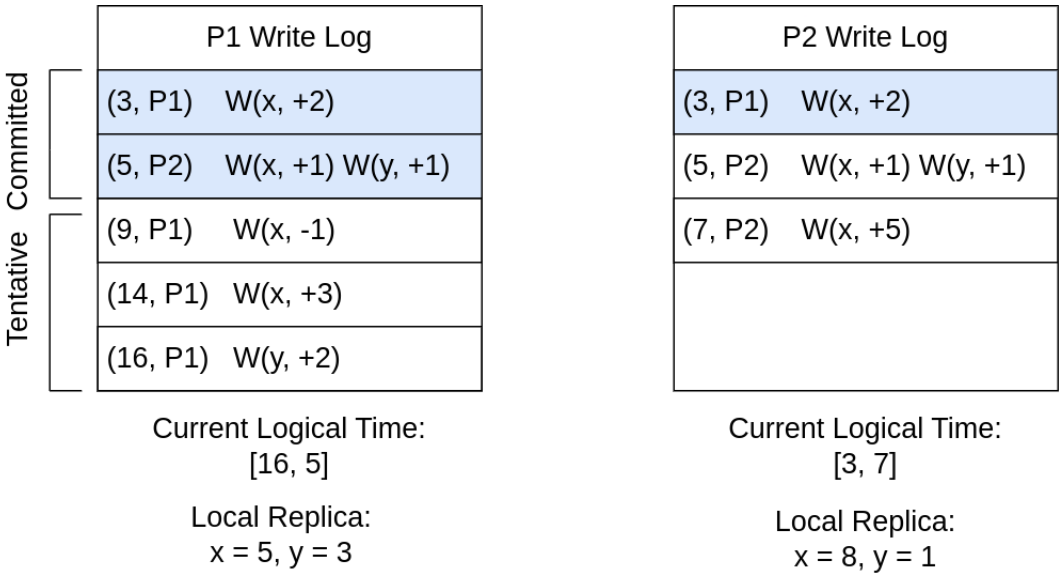


Figure 12: Snapshot of two local replicas using TACT

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.

5.3 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

5.4 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

5.5 Future work

6 Data Fusion

[19]

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.

6.1 Fusion centers

To be written.

6.2 Sheaf theory

6.2.1 Introduction to presheaves

Definition 6.1. 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 functions between the sets.

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

Definition 6.2. A *semiautomaton* is a monoid paired with a set.

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

Definition 6.3. A copresheaf is a *category acting on a family of sets*.

Definition 6.4. A presheaf is a *category acting covariantly on a family of sets*.

6.2.2 Introduction to sheaves

To be written.

6.2.3 The consistency radius

To be written.

7 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. C. D. of Forestry and F. Protection. Monument Fire.

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

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

14. M. Reardon. Post 9/11: Can we count on cell networks? *CBS News*.

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

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

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

18. United States Department of Agriculture. Standards for Airtanker Operations, 2019.

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

20. Wings Over the Rockies Air and Space Museum. Aerial Firefighting Progress — Behind the Wings on PBS.

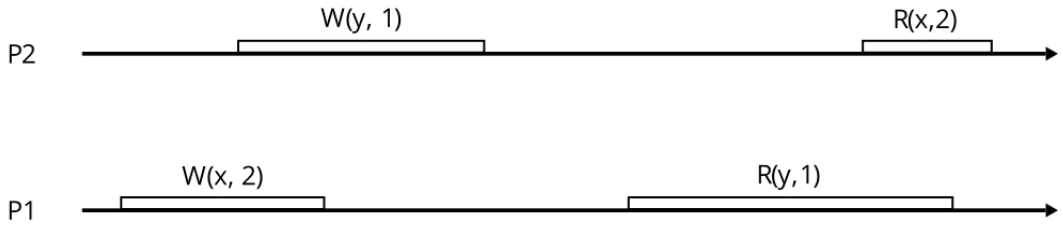
21. H. Yu and A. Vahdat. Building replicated internet services using tact: a toolkit for tunable availability and consistency tradeoffs. In *Proceedings Second International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems. WECWIS 2000*, pages 75–84, 2000.

22. 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.

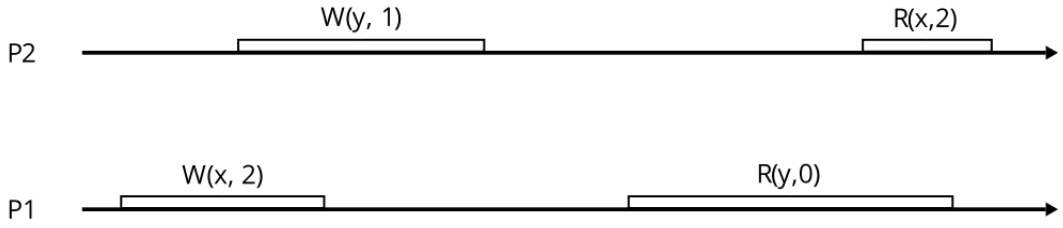
23. H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 123–133, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

24. 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.

25. 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.

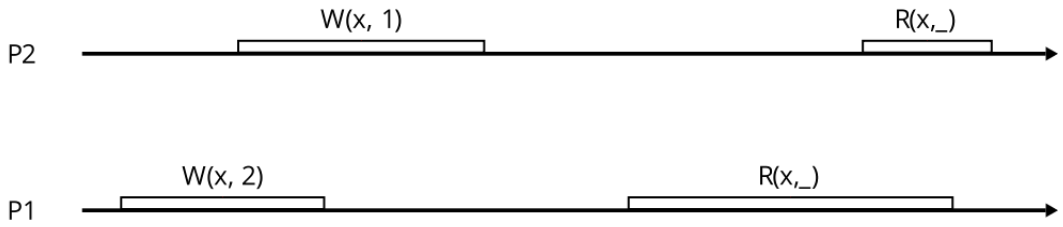


(a) A linearizable execution. Any choice of linearization works here.

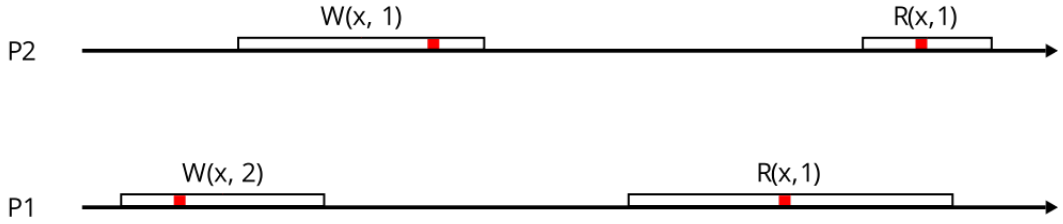


(b) A non-linearizable execution. The request to read y returns a stale value.

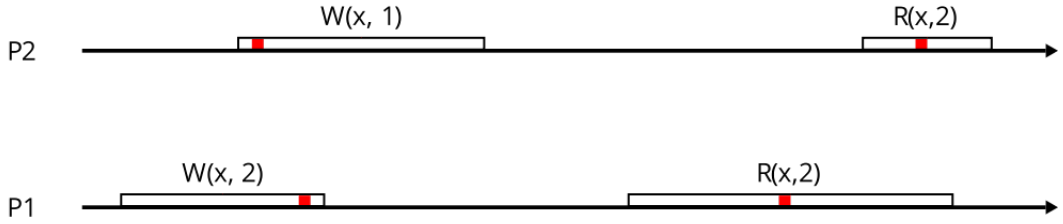
Figure 13: A linearizable and non-linearizable execution.



(a) An execution with read responses left unspecified.

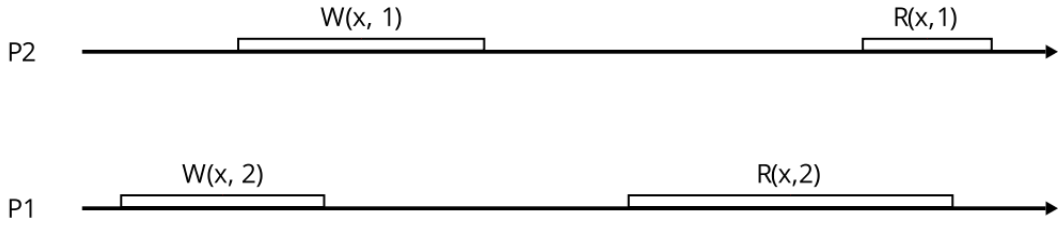


(b) A linearizable execution for which both reads return 1.

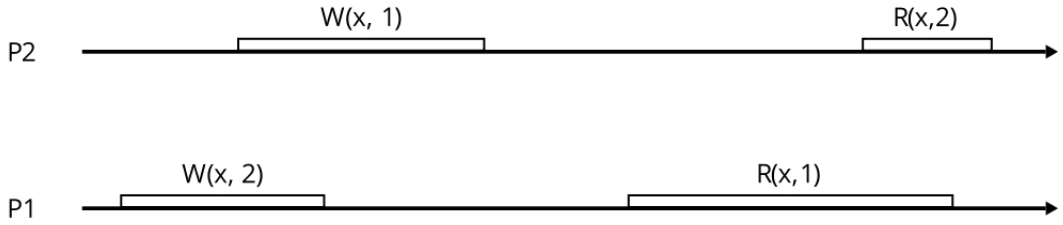


(c) A linearizable execution for which both reads return 2.

Figure 14: Two linearizable executions of the same underlying events that return different responses. Possible linearization points are shown in red.

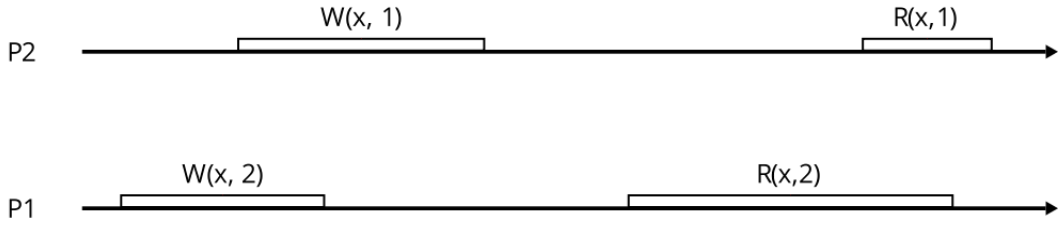


(a) A nonlinearizable execution with the read access returning disagreeing values. We will see later (Figure 16) that this execution is still sequentially consistent.

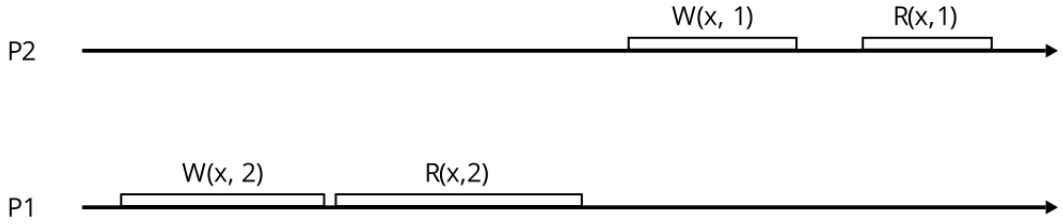


(b) Another nonlinearizable execution with read access values swapped. This execution is not sequentially consistent.

Figure 15: Two non-linearizable executions of the same events shown in Figure 14.

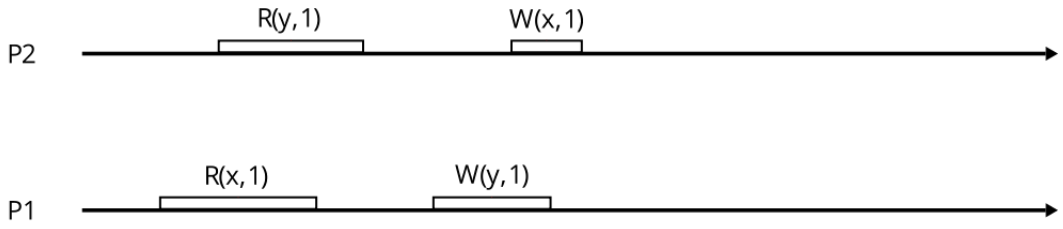


(a) A non-linearizable, sequentially consistent execution.

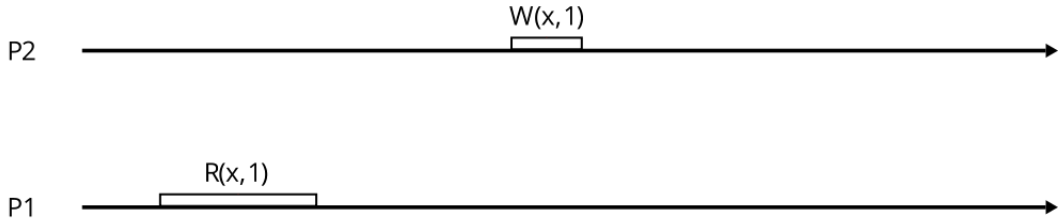


(b) An equivalent interleaving of 16a.

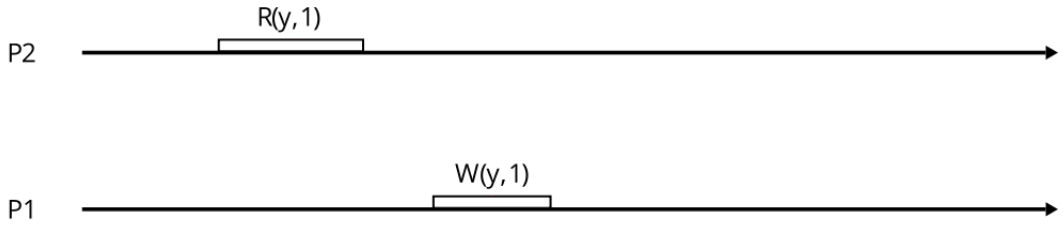
Figure 16: A sequentially consistent execution and a possible interleaving.



(a) A non-sequentially consistent execution.



(b) The sequentially consistent history of x.



(c) The sequentially consistent history of y.

Figure 17: A non-sequentially consistent execution with sequentially-consistent executions at each variable.

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