NASA/TM–2022–XXXXXX

Distributed Systems Challenges for Coordination of Airborne and Ground-Based Agents in Civil 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

Distributed Systems Challenges for Coordination of Airborne and Ground-Based Agents in Civil 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, taking disaster response scenarios as a motivating use case. Enforcing safety requirements for distributed agents requires coordination by passing messages over a communication network. However, the operating 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. This self-contained memo discusses some of the distributed systems challenges involved in system-wide safety, focusing on the practical shortcomings of both strong and weak consistency models for shared memory. Then we survey two *continuous* consistency models that come from different parts of the literature. Unlike weak consistency models, continuous consistency models provides hard upper bounds on the "amount" of inconsistency observable by clients. Unlike strong consistency, these models are flexible enough to accomodate real-world conditions, such as by providing liveness during brief network partitions or tolerating disagreements between sensors in a sensor network. We conclude that continuous consistency models are appropriate for analyzing safety-critical systems that operate without strong guarantees about network performance.

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 rapid growth in commercial aviation. To that end, the NASA Aeronautics' Airspace Operations and Safety Program (AOSP) System Wide Safety (SWS) project has been investigating new technologies and methods by which crewed and uncrewed aircraft may safely operate in shared airspace.

This memo surveys topics in distributed computing that are relevant to the challenge of maintaining system-wide safety. Our primary motivating use cases have been taken from civil emergency response scenarios, especially wildfire suppression and hurricane relief. The motivation for this choice is two-fold. First, the rules for operating in the US national airspace are typically relaxed during natural disasters and relief efforts. Second, these settings are an excellent microcosm for the sorts of general challenges faced by other, non-emergency applications. We summarize a range of diverse topics in the literature that may prove useful in overcoming these challenges.

A common albeit abstract theme is the notion of *continuity*, as in topology. Designing a system that is robust to real-world environments is an exercise in making tradeoffs, and it is of particular importance in this setting to exercise fine control over these tradeoffs. To build a system that is predictable—clearly a prerequisite for safety—changes in network conditions, usage patterns, etc. should generate (only) a proportional change in system behavior as observed by its users. In other words, the behavior of a safe system should in some sense be a continuous function of its inputs and operational environment.

1.1 Layout of this document

This document aims to be self-contained and accessible to a broad technical audience.

Section 2 starts with a pragmatic summary of disaster response and some of the relevant computing challenges in that setting. The rest of the document analyzes these issues further, proceeding from lower-level details (memory models and network protocols) to higher-level ones (database replication and data fusion).

Section 3 is an introduction to distributed systems, consistency, and a classic and fundamental result: the "CAP" theorem(s) for the atomic and sequential consistency models (Theorems 3.2 and 3.3, respectively). The CAP theorem is a negative result, proving that a distributed system cannot enforce strong consistency without going offline during network partitions. More generally, the result implies that maintaining strong consistency makes systemwide network performance an upper bound on the availability of a system to do useful work for clients, an unacceptable restriction. The practical importance of this result is that in realistic emergency response environments, agents will almost always act with less than perfect global information, a fact that will later motivate Section 6.

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

Section 5 examines networking considerations. Our vision of future emergency communication networks integrates concepts from delay/disruption-tolerant networks (DTN) and mobile ad-hoc networks (MANET) to provide digital communications that are robust to a turbulent operational environment. We also examine the state of software-defined networking (SDN), which allows networking equipment to be reprogrammed (e.g. to support new versions of protocols) without replacing the underlying hardware. Potential benefits of SDN include easier experimentation, reduced special-purpose infrastructure, rapid iteration, better interoperability between vendors, and better opportunities for simulation and formal verification of protocols.

Section 6 describes a hypothetical application of the sort that might be used in a disruption- or delay-heavy network: a data-replication service built on Yu and Vahdat's theory of *conits* (short for "consistency unit"). Unlike some shared memory frameworks, the conit framework provides neither idealized consistency nor guaranteed high-availability, but rather some quantifiable tradeoff between both ideals. In this sense it is a *continuous* model. The idea rests on the observation that many applications can tolerate inconsistency among replicas of a data item if a hard upper bound on the divergence between replicas can be assumed. A database replication middleware designed around the conit framework would allow developers to semantically define consistency units, enforce policies bounding application-defined measurements of (in)consistency between replicas of these units, and even dynamically tune these policies on the fly, say in response to network conditions or usage patterns.

Section 7 concerns data fusion. Now and in the future, agents in disaster scenarios will make decisions informed by many different kinds of data. This data may come from sensors in the environment, sensors attached to agents (e.g. a heartrate monitor CITE), information reported by agents and civilians, civil agencies (e.g. the U.S. National Weather Service), to name just a few. Efficient integration, processing, filtering, and dissimination of this data is necessary to avoid the familiar problem of "swimming in data and drowning in noise" (CITE). This task is especially challenging in our setting 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 arguably the "canonical" mathematical model for data fusion (CITE). Sheaf theory provides a rigorous framework for discussing how heterogeneous sources of noisy data can be integrated into a coherent picture, and can quantify and measure how well this task is achieved.

We conclude in Section 8 by recapping some of the main themes in this document and highlighting areas where design decisions must be made, ideally based on real-world experiments or simulations to assess system behavior under realistic conditions.

2 Coordination Challenges in Disaster Response

This section explains some aspects of disaster response scenarios that motivate the other sections of this document. We describe how real-world environments give rise to fundamental challenges that are best addressed with the sound application of distributed computing principles.

The operational environment of wildfire suppression, hurricane relief, and other disaster scenarios is generally characterized by systemic communications challenges. The causes of this fundamental reality are several in number: remote locations, difficult terrain, damaged infrastructure, harsh weather, and limited battery power, to name a few. Other complications follow from this basic fact, as it forces agents to choose between suffering long delays in sending or receiving information, or acting with only limited knowledge, or both. Both choices are problematic and can present safety challenges.

From a networking perspective, environmental and operational factors tend to cause agents (e.g. first responders) to experience heavy packet loss and significant delays in message-passing. This can happen at particularly inopportune times, e.g. when weather or fire conditions are at their worst, as in some cases the conditions which prompt urgent communication will correlate with those that make communication difficult.[1]

From a systems perspective, an unreliable network obstructs protocols for coordination among distributed agents, say to broadcast a message from one agent to the group. At a high level, any kind of coordinated action requires enforcing some notion of consistency across replicas of data shared between agents, e.g. the history of text messages in a chat application. Stronger notions of consistency generally require a greater ability to communicate with other agents quickly. Otherwise the system may have to wait for the network, delaying the processing of requests from clients in the meantime. In practical applications this implies that tradeoffs will have to be made *somewhere*, so it is necessary to understand where this happens and make choices informed by the particulars of the real environment.

2.1 Communication and Safety

An unreliable network is a safety challenge, as safety requires agents to act with up-to-date information about the world. Generally this information is received from sensors and other agents in the environment, who send it over the network(s), making the network a major bottleneck.

Consider firefighting airtankers for example. The largest examples of these, Very Large Airtankers (VLATs), can deposit more than 10,000 gallons of fire retardant at once, or more than enough to dislodge tree limbs—indeed, in some cases enough to crush a ground vehicle[2]). This creates a hazardous situation for ground crews. One potential policy would be to disallow this action if a airtanker's pilots do not have up-to-date information about the location of agents on the ground. However, shar-

[1] Consider that, tautologically, a communications network is most congested precisely when everyone needs to use it.

[2] Link

ing this information may be difficult or impossible if heavy smoke, a damaged radio tower, or a tall ridge prevents communications. In these scenarios, our hypothetical policy may prevent airtankers from operating, leading to potentially dangerous inefficiencies.

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

> $\mathbf{P}_{\text{safe}}$: All ground agents are known to be at least 300 meters 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 15 minutes of receiving a request.

Unfortunately, safety and liveness can be 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 them, which will be further explored in Section 3, means the two often cannot be guaranteed simultaneously. Such is the case in our example: if a group of ground agents is unable to broadcast their locations to the pilot, then the pilot's actions may have to be delayed to maintain $\mathbf{P}_{\text{safe}}$ at the cost of $\mathbf{P}_{\text{live}}$. This inaction could even come at the cost of allowing a dangerous fire to spread.[3]

We emphasize a point here: the issue in the previous example does not simply disappear if no ground personnel are actually within 300 meters of a drop zone. For example, perhaps firefighters were previously in the danger area and since left, so no danger is actually present, but the fact of them leaving is not conveyed to the pilot. To ensure $\mathbf{P}_{\text{safe}}$, an airtanker's actions must be restricted when pilots do not *know* whether an action would violate $\mathbf{P}_{\text{safe}}$—the knowledge of this fact, and not merely the fact of it, is the crucial part. Propagating knowledge between agents fundamentally requires communication, and communication is necessarily limited to what the available communication network(s) can provide. Thus, efficient use of a chaotic and congested network in these situations is paramount.

2.2 Communication in Practice

In the field, communication between firefighters and other agents in disaster response scenarios is often facilitated by handheld radios, which are inherently limited in their

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

battery life, bandwidth, effective range, and ability to work around environmental factors like foliage and smoke.

In our background research, we found an interview with a volunteer firefighter who relayed a story the Ironside repeater station, which was destroyed by fire in YEAR (CITE). This repeater had strategic importance, being located on a tall ridge, and its loss prevented communication between operators on different sides of the ridge, in other words creating a network partition. The partition continued until firefighters could ascend the ridge to deploy a temporary station, presumably diverting operators from other duties. This story demonstrates the potential for widespread system failure due to the loss of a central system component. It is also exactly the kind of scenario considered by Brewer's CAP theorem in Section 3.

In the field, it is common for firefighters to shout messages as an alternative to using a radio. Besides highlighting the fact that sometimes simple things work, there is a distributed computing lesson here, as it exemplifies a kind of "geospatial locality of reference" that system designers ought to take into account. We expect, generally speaking, that agents with a higher need to coordinate their actions will tend to be located closer to each other, which in turn correlates with an ability to communicate quickly and reliably. This kind of principle motivates the sort of decentralized, ad-hoc networking protocols considered in Section 5. It can also affect the design of higher-level applications like the one in Section 6.

Civil aviation has also traditionally employed simple communication patterns between airborne agents. 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 of aircraft may need to operate in a small area, near complex terrain, and at times operating at an altitude of less than 1000 ft. (CITE). In other words, the demands are many and the margins for error are small. This sort of use case demands more sophisticated coordination schemes between airborne and ground-based elements than solutions like ADS-B provide by themselves.

As aircraft generally have better line-of-site to ground crews than ground crews have to each other, firefighters sometimes relay messages to air-based units over radio, which in turn is relayed back down to other ground units. The locality principle comes into play here too, but this time in the unfortunate reverse direction. This simple relay scheme allows messages to travel farther, but the extended reach comes at the cost of introducing delays and possible degradation of message quality, as in the classic game of telephone.

A common principle in these examples is that communication over short distances is easy, so applications should take advantage of this fact when possible. Communication over long distances is harder. When the amount of communication demanded by clients exceeds what the network can actually provide, the system should be prudent in allocating scarce bandwidth to the most important messages.

2.3 Towards the Future

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

Taking a broader view, agents in disaster response environments will often be both producers and consumers of data, and this data will need to processed for agents to make informed decisions. Information shared by agents could include the following:

- The location of agents, vehicles, hazards, victims, etc.
- Free-form communication (e.g. voice or text messages)
- Medical information collected from e.g. digital triage tags (CITE)
- Data about current and predicted weather patterns
- Topographic information about the terrain
- Availability of resources, e.g. ambulances or firetankers on standby

In a perfect environment, such information would be shared with all necessary agents in whole and instantly. In reality, agents will be presented with information that is sometimes incomplete, out of date, or contradictory—all problems that are further exacerbated by an unreliable network.

At the network level, a future communication system should be opportunistic in exploiting every available opportunity to facilitate communication between agents in a transparent manner. For instance, an overhead drone surveying a fire may opportunistically act as a mobile base station (i.e. cell tower) to ground crews, routing digital messages through the network on their behalf, taking advantage of the drone's better ability to send and receive messages over long distances. In effect, this would be a high-tech modernization of the sort of informal relay scheme operating today over traditional radio channels.

3 Introduction to Distributed Systems

A distributed system, in the most general sense, is a collection of independent entities that cooperate to solve a problem that cannot be individually solved (Kshemkalyani and Singhal 2008). (Singhal and Shivaratri 1994) 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."

In our scenarios, system nodes typically represent computers, routers, sensors, and communication devices, while the clients would typically be firefighters using these devices and other persons involved in disaster response efforts. The components of the system collectively accomplish goals such as navigating safely in close proximity, delivering resources to remote locations, and suppressing fires.

A fundamental goal for distributed computing systems is to "[appear] to the users of the system as a single coherent computer" (Tanenbaum and Steen 2007). This can be understood as the requirement that all nodes present a *mutually-consistent* view of the world, e.g. the state of a globally-maintained database, to system clients. Bad things happen when consistency is violated:

- A bank client would be unhappy if deposits that appear in their account online are not reflected when they check their balance at an ATM, or if they seem to disappear after refreshing the webpage.
- If two air traffic controllers were presented with conflicting or information about the trajectory of aircraft, they could potentially issue dangerously incorrect instructions to the pilots.
- Resource-tracking systems are not useful if a resource that appears to be available cannot actually be used because the information is out of date. Alternatively, a resource that is actually available may not be used if clients think it is still unavailable.

In general, violating consistency means the abstraction of a single shared universe is broken. In extreme cases, this can invalidate clients' mental model of the system, make the system's behavior harder to predict, or cause safety requirements to be violated. Clearly, all other things being equal, one wants to have as much consistency as possible.

When *strong* notions of consistency are enforced, clients are presented with the abstraction of a single shared world, i.e. as if they are all connected to a central computer rather than a complex system of independent computers. This abstraction shields clients and application developers from complexity and makes it simpler to reason about a system's behavior. However, we shall see that strong notions of consistency are brittle in the sense that they generally cannot be achieved for the kinds of systems we consider in this document.

3.1 Message Passing

A distributed system consists of a set $\mathcal{P} = \{P_i\}_{i \in I}$ of *processes*, which we think of these as executing on independent, often geographically dispersed computers that communicate by message-passing. Processes can *only* coordinate by passing messages over the network.[^fn] [^fn]: This fact is implied by absence of a common memory, whereas processes on the same machine have the option to share data by writing it to a memory location both processes have access to.

A foundational assumption is that the network is almost always less than perfectly reliable—this fact can be counted on during emergencies. Imperfection means that message delivery is not instantaneous may be unpredictable. The network may deliver a packet zero times (i.e. it may silently delete the packet) or multiple times, and furthermore different packets may arrive in any order.[4] All of these behaviors represent obstructions to consistency and make it challenging to enforce safety requirements. We shall now make this discussion more precise.

3.1.1 Client-server architecture

Process take requests from clients, such as to read or write a value in a database. The lifecycle of a typical request is depicted in Figure 1. At some physical time (i.e. wall-clock time) $C.s \in \mathbb{R}$ (client start time), a client sends a message to a process. At time $E.s$, which we'll call the *start time* of the event, the message is accepted by the process. The request is processed until some strictly greater time $E.t > E.s$ when a response is sent back to the client. The value $E.t - E.s$ is the *duration* of the event.

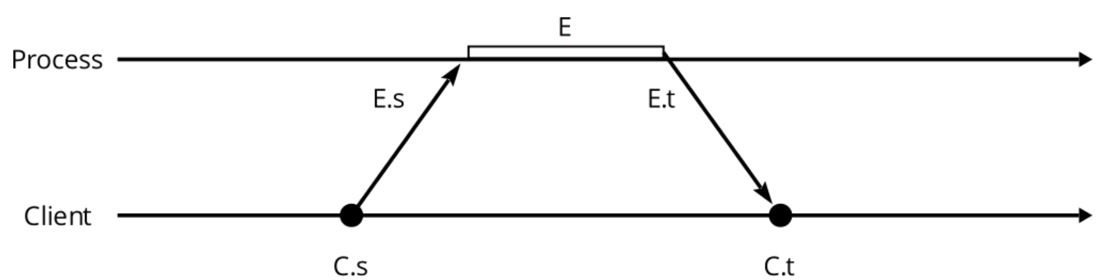


Figure 1: Lifetime of a client request

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

[4] In the case of *Byzantine faults* (CITE) the network may even act like a malicious adversary, though we will not consider this scenario in this document.

3.1.2 Measuring time

3.1.3 Causal order/external order

As is often the case, we shall assume that requests handled by a single process do not overlap in time. This can be enforced with local serialization methods such as two-phase locking (CITE) that can be used to isolate concurrent transactions from each other, providing the abstraction of a system that handles requests one at a time. On the other hand, any two processes may handle two events at the same physical time, so that there is no obvious total order of events across the system. Instead, one has a partial order called *external order*. Intuitively, it is the partial order of events that would be witnessed by an observer recording the real time at which systems begin and finish responding to requests.

Definition 3.1. Let E be an execution. Request $E1$ *externally precedes* request $E2$ if $E1.t < E2.s$. That is, if the first request terminates before the second request is accepted. This induces an irreflexive partial order called *external order*.

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

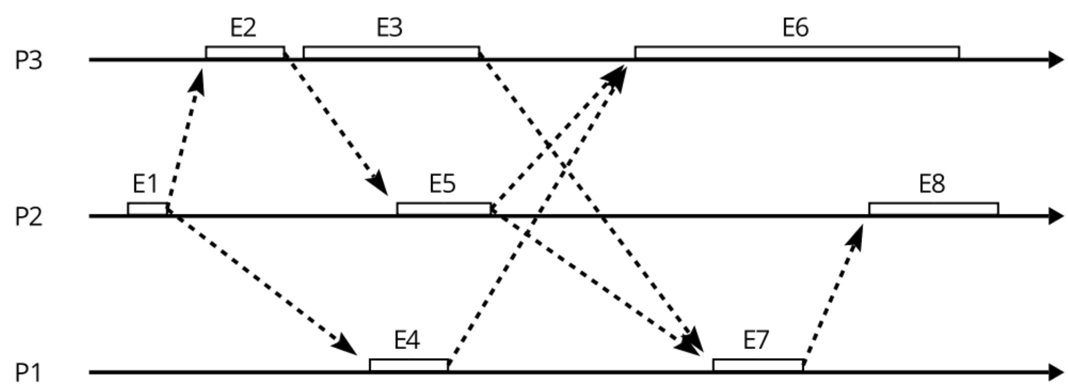
Because we assume processes handle events one-at-a-time, the events handled at any one process are totally ordered by external order—one event cannot start before another has finished. If $E1$ and $E2$ are events at *different* processes, they need not be comparable by external order, i.e. neither $E1.t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

Definition 3.2. If two events overlap in physical time (equivalently, if they are not comparable by external order), we call the events *physically concurrent* and write $E1||E2$.

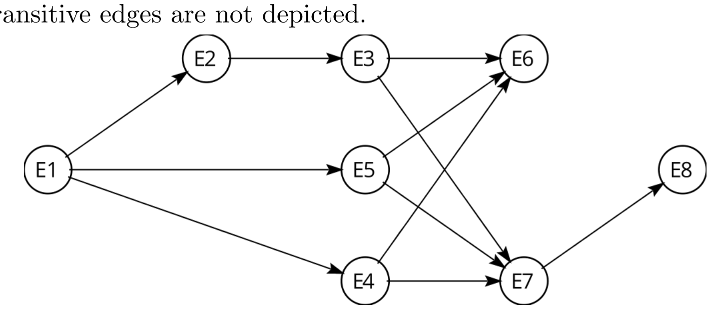
Physical concurrency is a reflexive and symmetric—but usually not transitive—binary relation. Such structures are often called *compatibility relations.* The general intuition is that anything is compatabile with itself (reflexivity), and the compatibility of two objects does not depend on their order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

Figure 2a shows the external order relation for an execution. To save space we elide arrows between two events of the same process and arrows that can be inferred by transitivity. This corresponds to the directed acyclic graph structure shown in 2b.

The reader may wonder if we can consider events to be totally ordered, say by pairing them with a timestamp that records their physical start time to resolve ties like $x||y$. This order is not generally useful for a couple reasons. First, we assume processes have only loosely synchronized clocks, so timestamps from two different processes may not be comparable. Additionally, even systems that enforce linearizable consistency (c.f. Section 3.2) do not necessarily handle requests in order of their physical start times.



(a) Depiction of external order between concurrent events across three processes. Intra-process and transitive edges are not depicted.



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

Figure 2: External order

3.2 Memory Consistency Models

What precisely constitutes consistency? One can choose from multiple consistency models, and the most appropriate model depends on the semantics expected by the application and its clients, which must be weighed against other requirements.

To discuss consistency models, we shall be less interested in the details of message-passing and more interested just in the responses observed by clients. We shall consider the full set of events across a distributed system, such as shown in Figure 2. This is called an *execution*. Consistency models constrain the set of allowable return values in response to clients' requests.

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

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

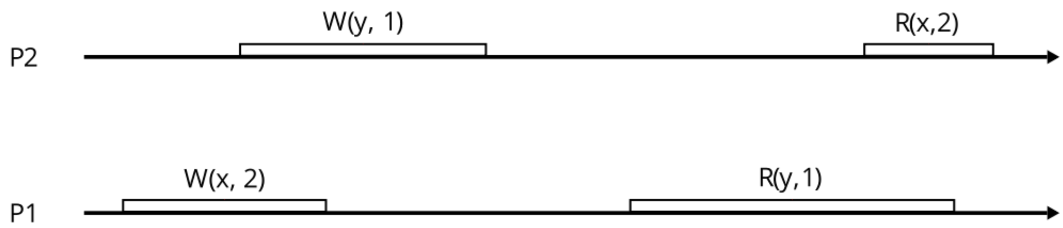
A *memory consistency model* formally constrains the allowable system responses during executions. *Strong* consistency models are generally understood as ones provide the illusion that all clients are accessing just one globally shared replica. As we will see, this still leaves room for different possible behaviors (i.e. allows non-determinism in the execution of a distributed application), but the allowable behavior is tightly constrained.

3.2.1 Linearizability/atomic consistency

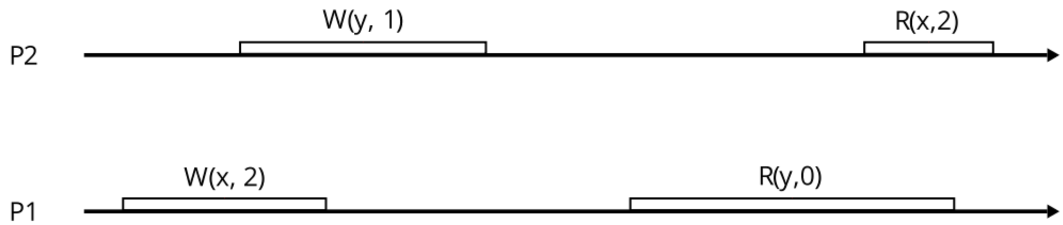
Linearizability (Herlihy and Wing 1990) is essentially the strongest common consistency model. It is known variously as atomic consistency, strict consistency, and sometimes external consistency. In the context of database transactions (which come with other guarantees, like isolation, that are more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

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

Figure 3a 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



(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 3: A linearizable and non-linearizable execution.

took effect. If client on P_1 read a stale value, we would say the execution is not linearizable. Figure 3b shows an non-linearizable execution that returns stale data instead of reflecting the write access to y on $P2$.

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

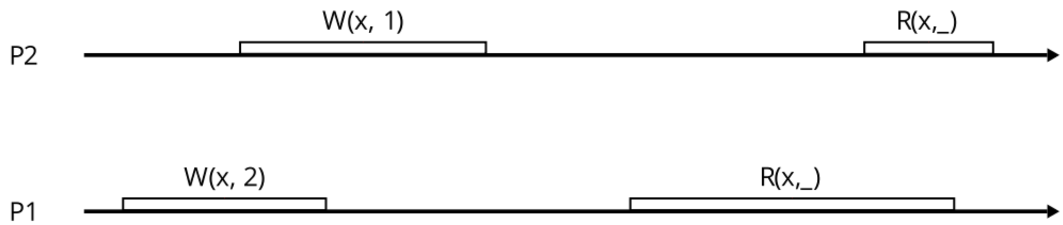
Definition 3.3. A *linearization point* $t \in \mathbb{R} \in [E.s, E.t]$ for an event E is a time between the event's start and termination. An execution is *linearizable* if and only if there is a choice of linearization point for each access, which induces a total order called a *linearization*, such that E is equivalent to the serial execution of events when totally ordered by their linearization points.

Linearization points are demonstrated in Figure 4. 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 5).

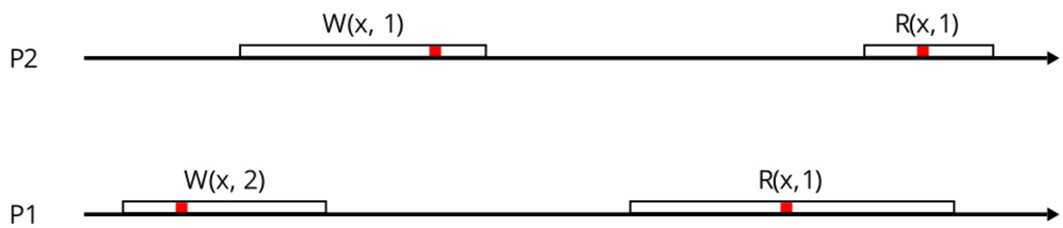
Definition 3.4. We say an entire system is linearizable when all possible executions of the system are linearizable.

3.2.2 Sequential consistency

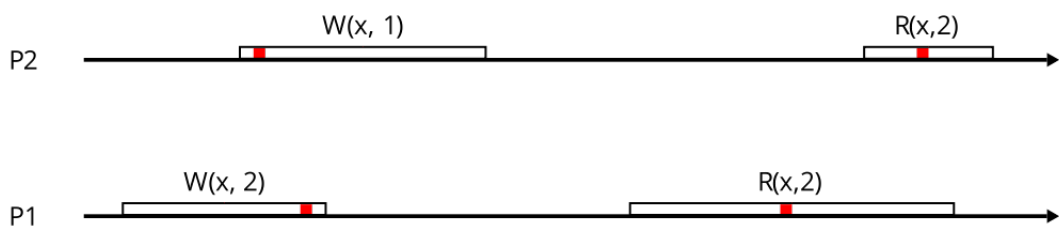
Enforcing atomic consistency means that an access E at process P_i cannot return to the client until every other process has been informed about E. For many applications this is an unacceptably high penalty. A weaker model that is still strong



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

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, 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 3.5. A *sequentially consistent* execution is characterized by three features:

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

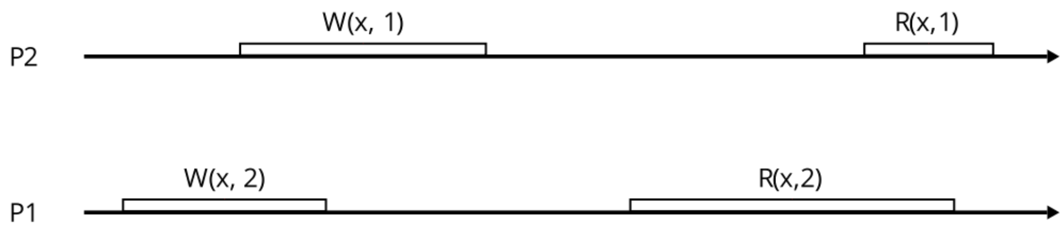
Processes in a sequentially consistent system are required to agree on a total order of events, presenting the illusion of a shared database from an application programmer's point of view. However, this order need not be given by external order. Instead, the only requirement is that sequential history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

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

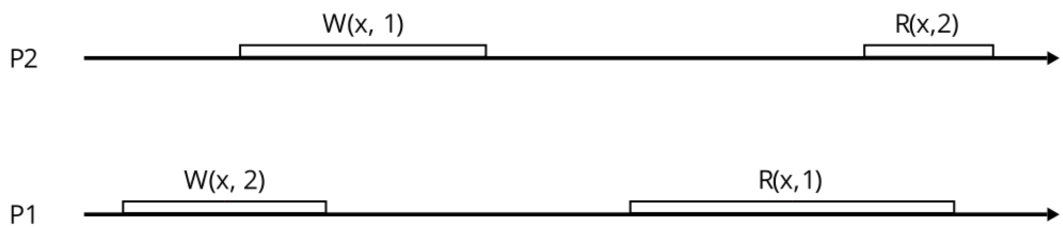
Proof. This follows because process order is a subset of external order. □

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

The converse of Lemma 3.1 does not hold. For example, Figure 6a was previously shown (Figure 5a) as a nonlinearizable execution. However, it is sequentially consistent, as evidenced by the interleaving in Figure 6b that slides the events $W(x, 1)$ and $R(x, 2)$ past each other.

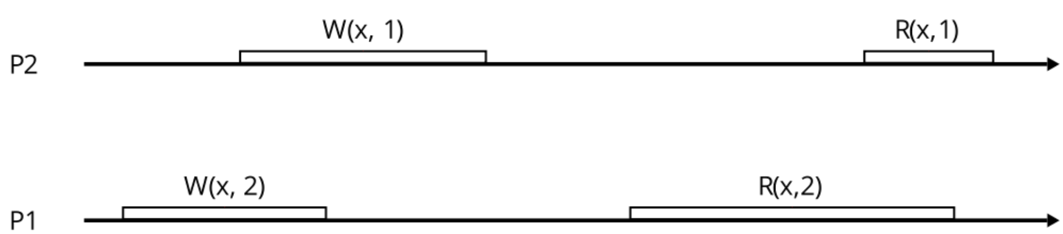


(a) A nonlinearizable execution with the read access returning disagreeing values. We will see later (Figure 6) that this execution is still sequentially consistent.

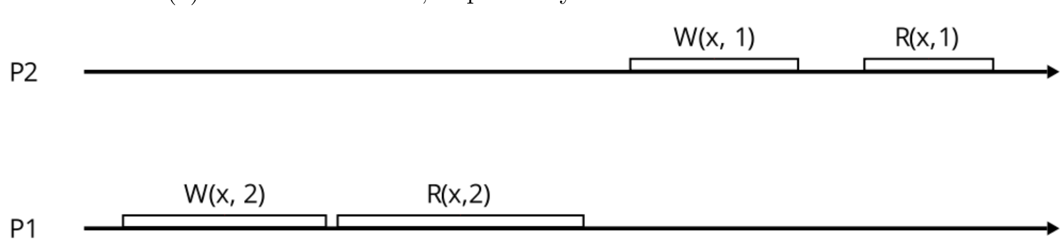


(b) Another nonlinearizable execution with read access values swapped. This execution is not sequentially consistent.

Figure 5: Two non-linearizable executions of the same events shown in Figure 4.

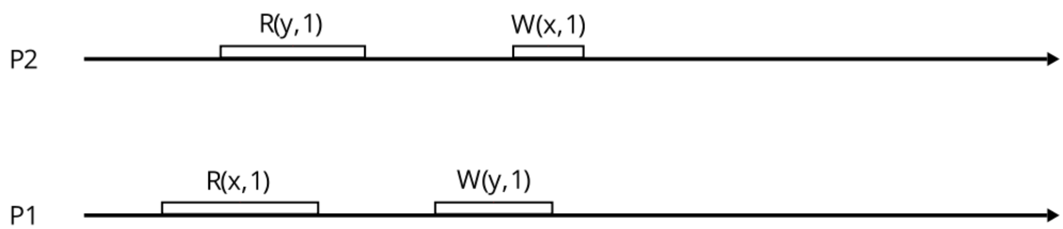


(a) A non-linearizable, sequentially consistent execution.

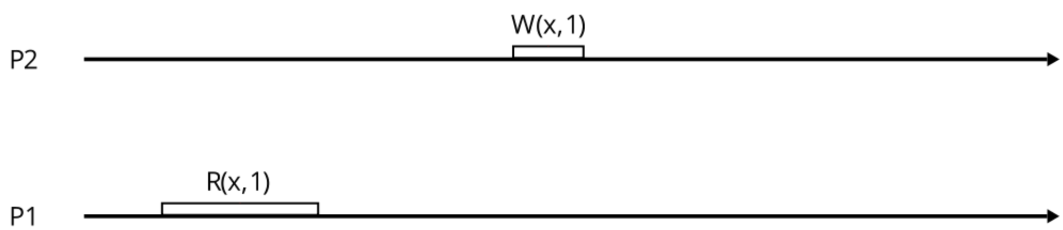


(b) An equivalent interleaving of 6a.

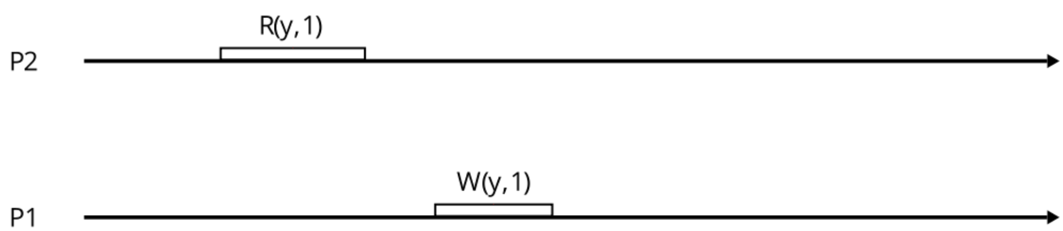
Figure 6: A sequentially consistent execution and a possible interleaving.



(a) A non-sequentially consistent execution.



(b) The sequentially consistent history of x.



(c) The sequentially consistent history of y.

Figure 7: A non-sequentially consistent execution with sequentially-consistent executions at each variable.

3.2.3 Causal consistency

Suppose in a message stream two questions Q_1 and Q_2 are asked and responses A_1 and A_2 are given, respectively. If the answers arrive in the wrong order, i.e. A_2 appears as the answer to Q_1 and vice versa, the potential for confusion can be severe.

3.3 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.

3.3.1 Safety/Liveness

The tension between consistency and availability is a prototypical example of a deeper tension in computing: that between safey and liveness properties (Davidson, Garcia-Molina, and Skeen 1985; Gilbert and Lynch 2012). These terms can be understood as follows.

- **Safety** properties ensure that a system avoids doing something "bad" like violating a consistency invariant. Taken to the extreme, one way to ensure safety is to do nothing. For instance, we could enforce safety by never responding to read requests in order to avoid offering information that is inconsistent with that of other nodes.

- **Liveness** properties ensure that a system will eventually do something "good", like respond to a client. Taken to the extreme, one very lively behavior would be to immediately respond to user requests, without taking any steps to make sure this response is consistent with that of other nodes.

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

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

3.3.2 The Cap Theorem

Fox and Brewer (Fox and Brewer 1999) 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 (Gilbert and Lynch 2002). The theorem is often somewhat misunderstood, as we discuss, so it is worth clarifying the terms used.

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

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

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

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

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

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

In Figure 3a, suppose the two processes are on opposite sides of a network partition, so that no information can be exchanged between them (even indirectly through a third party). If we just consider the execution of P_2 by itself, without P_1, linearizability would require it to read the value 2 for y. If we do consider P_1, linearizability requires that the read access to y must return 1. But if P_2 cannot send messages to P_1, then P_2's behavior cannot be influenced by the write access to y, so it would still have to return 2, violating consistency. Alternatively, it could delay returning any result until it is able to exchange messages with P_1. But if the partition never recovers, P_1 will wait forever, violating availability. □

3.3.3 CAP for sequential consistency

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

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

Proof. The proof is an adaptation of Theorem 3.2. Suppose P_1 and P_2 form of CAP-available distributed system and consider the following execution: P_1 reads x, then assigns y the value 1. P_2 reads y, then assigns x the value 1. (Note that this is the sequence of requests shown in Figure 7a, 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. □

As discussed in (Muñoz-Escoí et al. 2019), this stronger theorem was essentially proved by Birman and Friedman (Friedman and Birman 1996), before the CAP theorem.

3.3.4 Interpretation

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 (Brewer 2012). 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.

4 Desiderata

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

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

4.0.1 D1: Quantifiable bounds on inconsistency

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

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

4.0.2 D2: Accommodation of heterogeneous nodes

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

One can expect a variety of hardware in the field. For example, wildfires often involve responses from many different fire departments, and it must be assumed that they are not always using identical systems. Different participants in the system may be solving different tasks, with different levels of access to the network, and they present different risks. With these sorts of factors in mind, one should hope for frameworks that are as general as possible to accomodate a wide variety of clients.

4.0.3 D3: Optimization for a geodistributed wide area network

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

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

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

5 Resilient Network Architectures

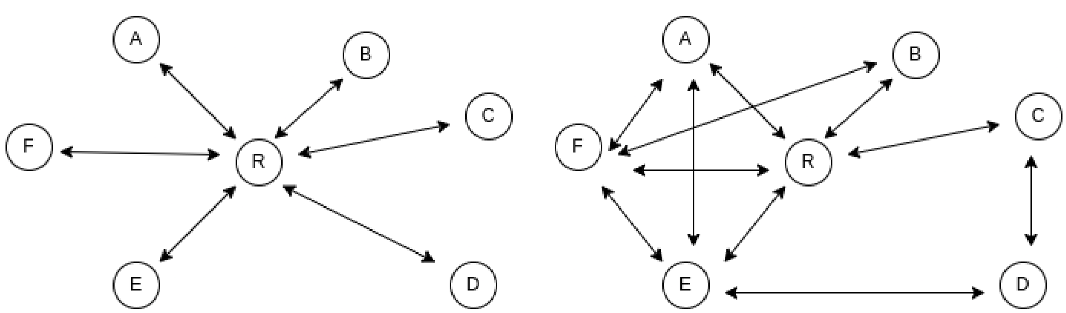
5.1 Ad-hoc networking

5.1.1 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 6. For such applications, it may be important to optimize for the sorts of usage patterns encountered in real scenarios, which are affected by (among other things) the low-level details of the network.

The *celluar* model (Figure 8a) 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 8b) assumes nodes communicate by passing messages directly to each other. This requires nodes to maintain information about things like routing and the approximate location of other nodes in the system, increasing complexity and introducing a possible source of inconsistency. However, it may be more workable given (i) the geographic mobility of agents in our scenarios (ii) difficult-to-access locations that prohibit setting up communication towers (iii) the inherent need for system flexibility during disaster scenarios.



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

Figure 8: Network topology models for geodistributed agents. Edges represent communication links (bidirectional for simplicity).

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

5.2 Delay-tolerant networking

5.3 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.

5.4 Software-defined networking

5.5 Verification of networking protocols

6 Continuous Consistency

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

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

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

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

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

As a third example,

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

Yu and Vahdat explored the CAP tradeoff from this perspective in a series of papers (Yu and Vahdat 2000a, 2000c, 2000b, 2001, 2002). They propose a theory of *conits*, a logical unit of data subject to their three metrics for measuring consistency. By controlling the threshold of acceptable inconsistency of each conit as a continuous quantity, applications can exercise precise control the tradeoff between consistency and performance, trading one for the other in a gradual fashion.

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

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

- The tradeoff of CAP is a continuous spectrum between linearizability and high-availability. More importantly, it can be tuned in real time.

- TACT captures neither CAP-consistency (i.e. neither atomic nor sequential consistency) nor CAP-availability (read and write requests may be delayed indefinitely if the system is unable to enforce consistency requirements because of network issues).

6.1 Causal consistency

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

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

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

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

Lemma 6.1. *Sequential consistency implies causal consistency.*

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

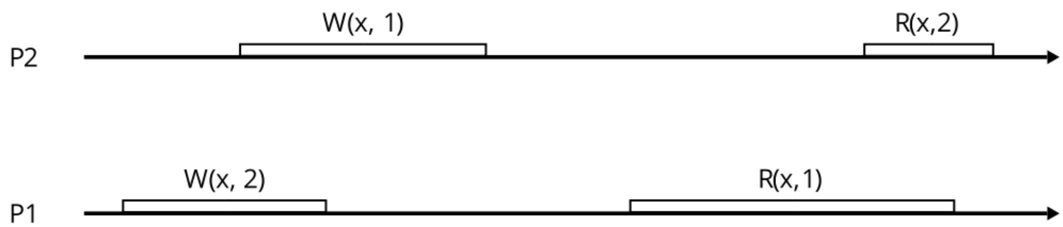


Figure 9: A causally consistent, non-sequentially-consistent execution

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

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

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

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

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

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

6.2 TACT system model

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

However, access to the data store is mediated by a middleware library, which sits between the local copy of the replica and the client. At a high level, TACT will allow an operation to take place if it does not violate user-specific consistency bounds. If allowing an operation to proceed would violate consistency constraints, the operation blocks until TACT synchronizes with one or more other remote replicas. The operation remains blocked until TACT ensures that executing it would not violate consistency requirements.

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

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

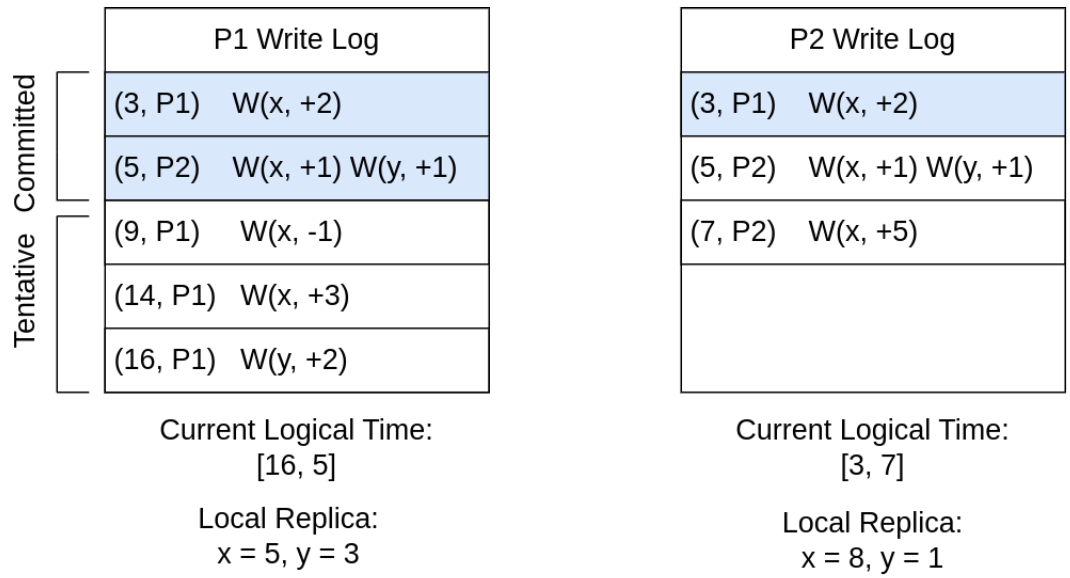


Figure 10: Snapshot of two local replicas using TACT

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

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

6.3 Measuring consistency on conits

Numerical consistency

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

Real time consistency

6.4 Enforcing inconsistency bounds

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

Order consistency

Real time consistency

6.5 Future work

7 Data Fusion

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

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

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

7.1 Fusion centers

To be written.

7.2 Sheaf theory

7.2.1 Introduction to presheaves

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

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

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

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

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

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

7.2.2 Introduction to sheaves

To be written.

7.2.3 The consistency radius

To be written.

8 Conclusion

To be written

Bibliography

Brewer, Eric. 2012. "CAP Twelve Years Later: How the "Rules" Have Changed." *Computer* 45 (2): 23–29. https://doi.org/10.1109/MC.2012.37.

Davidson, Susan B., Hector Garcia-Molina, and Dale Skeen. 1985. "Consistency in a Partitioned Network: A Survey." *ACM Comput. Surv.* 17 (3): 341–70. https://doi.org/10.1145/5505.5508.

Fox, A., and E. A. Brewer. 1999. "Harvest, Yield, and Scalable Tolerant Systems." In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 174–78. https://doi.org/10.1109/HOTOS.1999.798396.

Friedman, Roy, and Ken Birman. 1996. "Trading Consistency for Availability in Distributed Systems." USA: Cornell University.

Gilbert, Seth, and Nancy A. Lynch. 2002. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." *SIGACT News* 33 (2): 51–59. https://doi.org/10.1145/564585.564601.

———. 2012. "Perspectives on the CAP Theorem." *Computer* 45 (02): 30–36. https://doi.org/10.1109/MC.2011.389.

Herlihy, Maurice P., and Jeannette M. Wing. 1990. "Linearizability: A Correctness Condition for Concurrent Objects." *ACM Trans. Program. Lang. Syst.* 12 (3): 463–92. https://doi.org/10.1145/78969.78972.

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

Muñoz-Escoí, Francesc D, Rubén de Juan-Marín, José-Ramón García-Escrivá, J R González de Mendívil, and José M Bernabéu-Aubán. 2019. "CAP Theorem:

Revision of Its Related Consistency Models." *The Computer Journal* 62 (6): 943–60. https://doi.org/10.1093/comjnl/bxy142.

Singhal, Mukesh, and Niranjan G. Shivaratri. 1994. *Advanced Concepts in Operating Systems*. USA: McGraw-Hill, Inc.

Tanenbaum, Andrew S., and Maarten van Steen. 2007. *Distributed Systems: Principles and Paradigms*. 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall.

Yu, Haifeng, and Amin Vahdat. 2000a. "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*, 75–84. https://doi.org/10.1109/WECWIS.2000.853861.

———. 2000b. "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: USENIX Association.

———. 2000c. "Efficient Numerical Error Bounding for Replicated Network Services." In *Proceedings of the 26th International Conference on Very Large Data Bases*, 123–33. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

———. 2001. "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*, 429–38. IEEE Computer Society. https://doi.org/10.1109/ICDSC.2001.918973.

———. 2002. "Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services." *ACM Trans. Comput. Syst.* 20 (3): 239–82. https://doi.org/10.1145/566340.566342.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)*
01-12-2022

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Distributed Systems Challenges for Coordination of Airborne and Ground-Based Agents in Civil 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, taking disaster response scenarios as a motivating use case. Enforcing safety requirements for distributed agents requires coordination by passing messages over a communication network. However, the operating 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. This self-contained memo discusses some of the distributed systems challenges involved in system-wide safety, focusing on the practical shortcomings of both strong and weak consistency models for shared memory. Then we survey two *continuous* consistency models that come from different parts of the literature. Unlike weak consistency models, continuous consistency models provides hard upper bounds on the "amount" of inconsistency observable by clients. Unlike strong consistency, these models are flexible enough to accomodate real-world conditions, such as by providing liveness during brief network partitions or tolerating disagreements between sensors in a sensor network. We conclude that continuous consistency models are appropriate for analyzing safety-critical systems that operate without strong guarantees about network performance.

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

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

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18