Distributed Shared Memory and Remote Procedure Calls

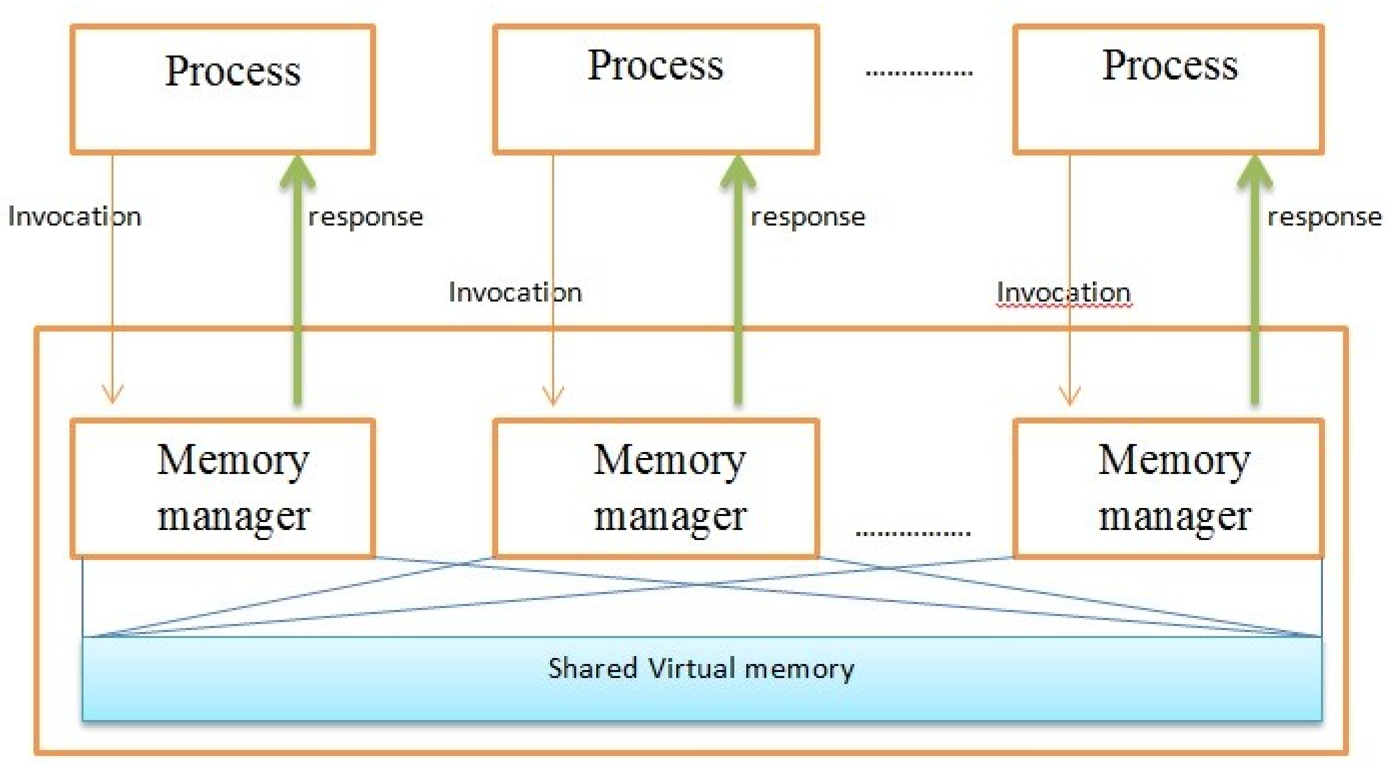
CSE 536 Spring 2026

jedimaestro@asu.edu

Outline

- Distributed Shared Memory
  - Consistency models
- Remote Procedure Calls

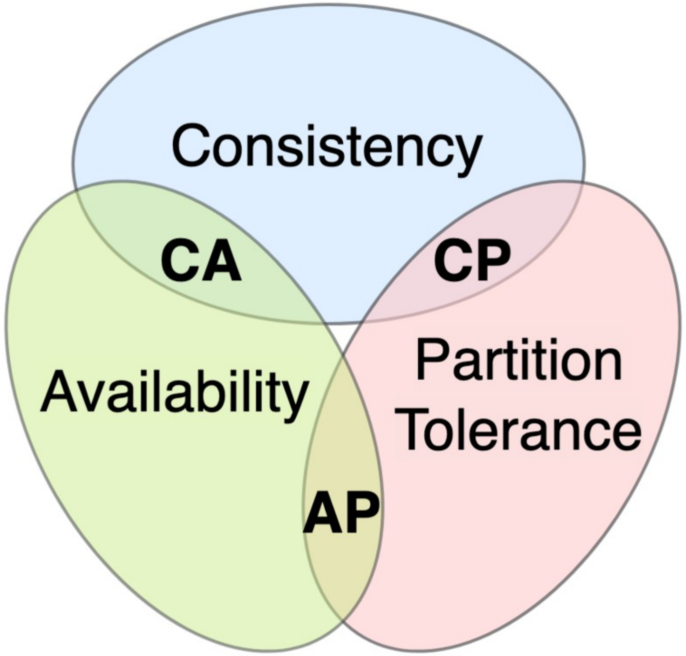
https://en.wikipedia.org/wiki/Distributed_shared_memory



Distributed shared memory

But, we need some notion of consistency...

https://en.wikipedia.org/wiki/CAP_theorem



Consistency – every read sees the most recent write, or an error.

Availability – Every request received by a non-failing node in the system must result in a response (not necessarily most recent).

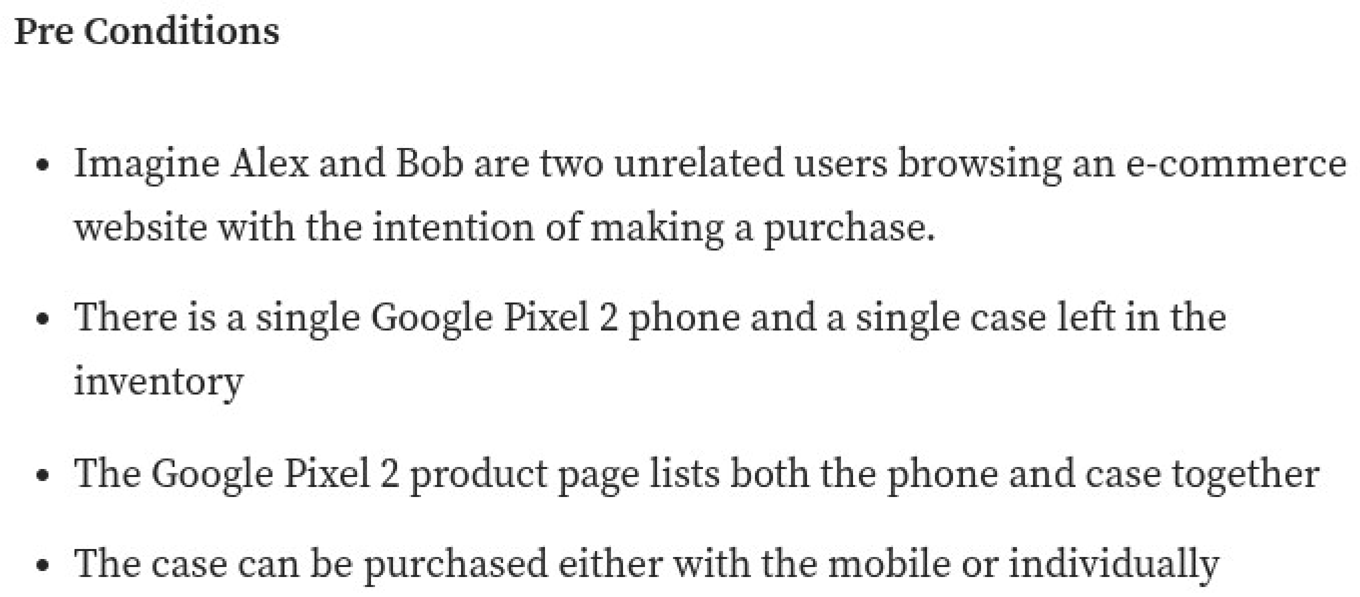
Partition tolerance – The system continues to operate despite an arbitrary number of messages being dropped (or delayed).

Two choices when failures happen

- Cancel the operation, at the expense of availability
- Proceed, but risk inconsistency

Strict consistency...

https://shashankbaravani.medium.com/consistency-models-in-distributed-systems-d64c134f84ce



Pre Conditions

- Imagine Alex and Bob are two unrelated users browsing an e-commerce website with the intention of making a purchase.
- There is a single Google Pixel 2 phone and a single case left in the inventory
- The Google Pixel 2 product page lists both the phone and case together
- The case can be purchased either with the mobile or individually

Strict consistency

(Read)R1 Alex arrives on Google Pixel 2 product page. He sees “Hurry up. Only 1 left!”

[**R2**]. Bob arrives on mobile case product page. He sees “Hurry up. Only 1 left!”

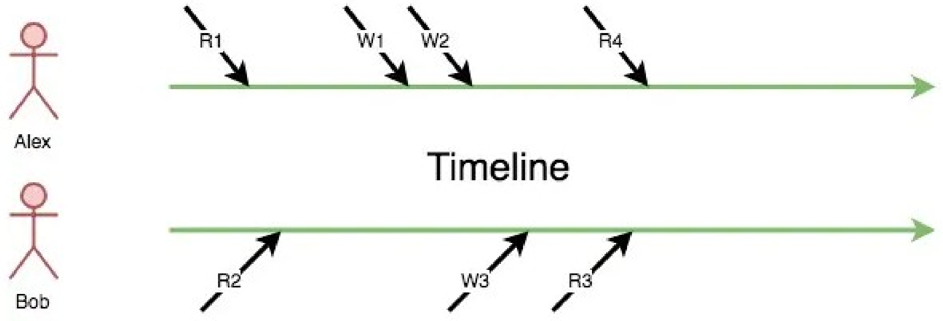
(Write)[W1] & [**W2**] Alex selects both the phone and the case and clicks on pay button. An update request is fired to first “block” the the mobile phone and then conditionally block the mobile case

[**W3**] Bob selects the case and clicks on pay button

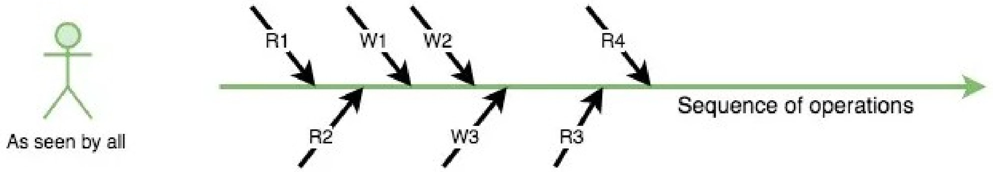
[**R3**] Bob’s blocking of mobile case is unsuccessful and he sees “Sorry the item is unavailable”.

[**R4**] Alex’s blocking of mobile phone and the mobile case are successful. Alex sees "Proceed to pay"

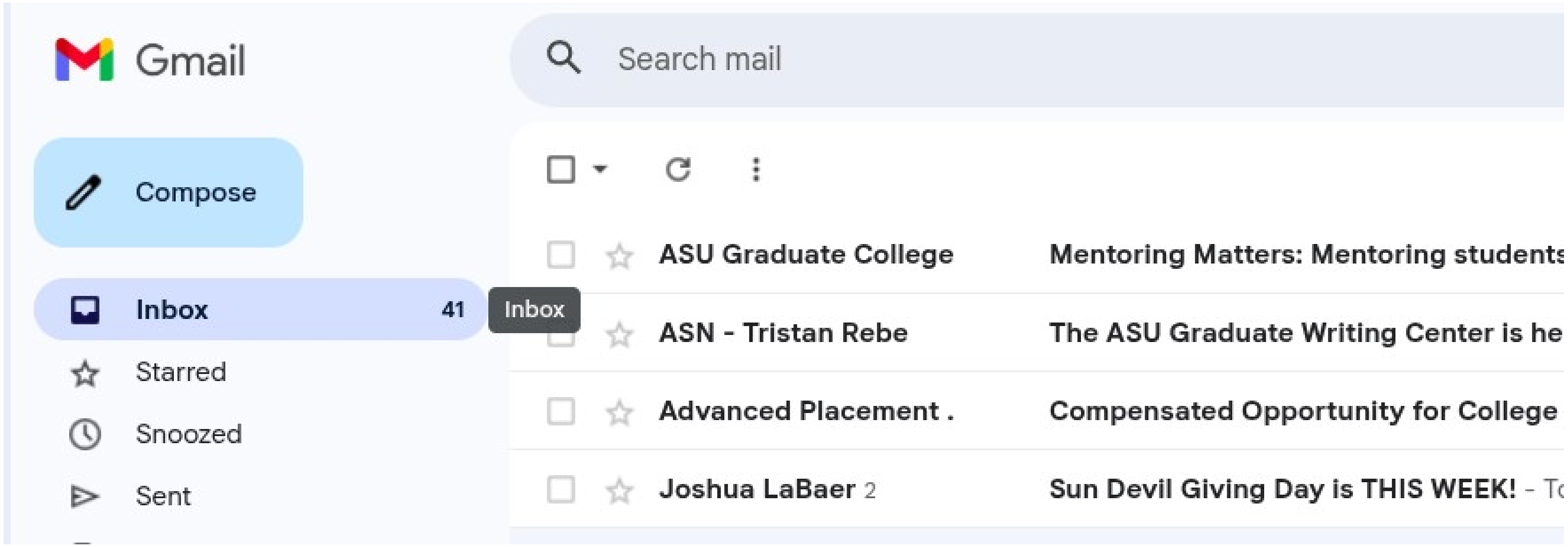
Strict consistency



The *total order* of operations i.e. [R1, R2, W1, W2, W3, R3, R4] is preserved for both Alex and Bob



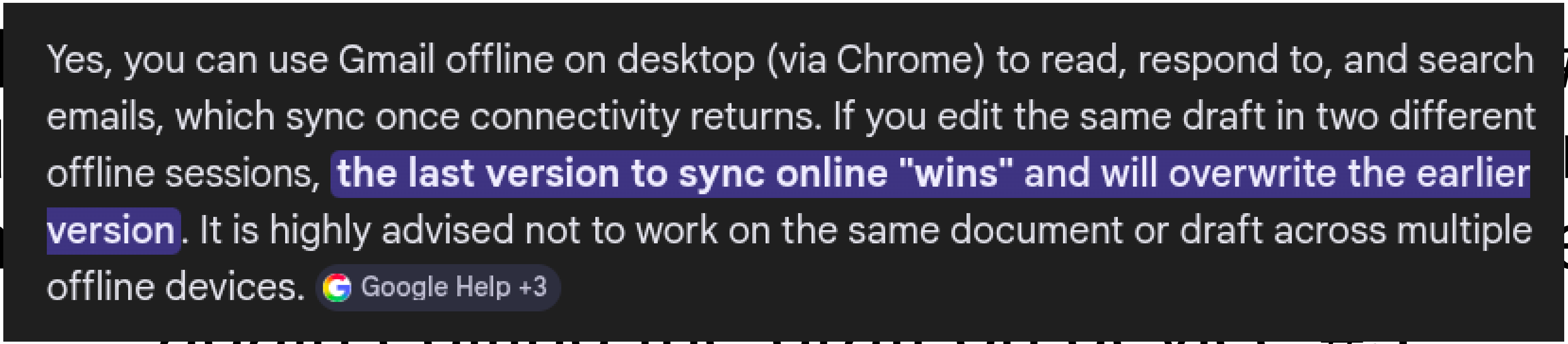
Example: Gmail (spanner)



ASU Graduate College Mentoring Matters: Mentoring students
ASN - Tristan Rebe The ASU Graduate Writing Center is he
Advanced Placement . Compensated Opportunity for College
Joshua LaBaer 2 Sun Devil Giving Day is THIS WEEK! - To

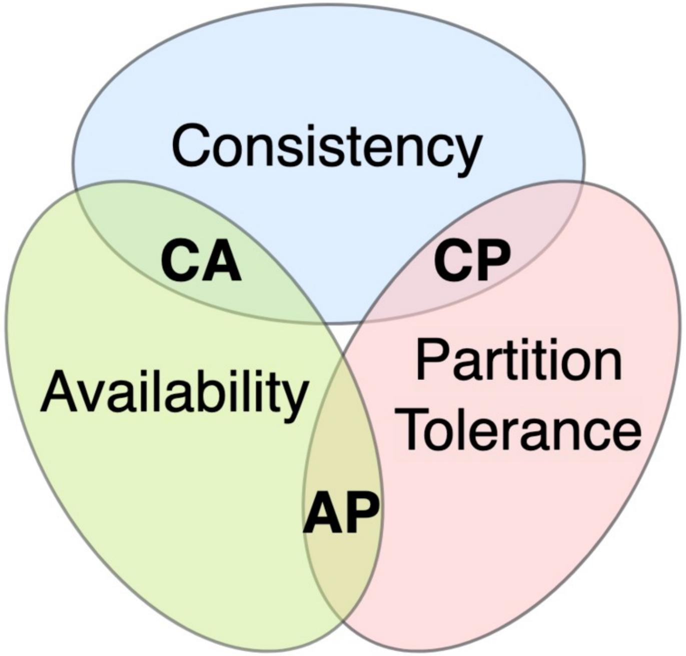
What happens if you create a draft on device #1, edit the draft on device #2 but right after opening the draft the device goes offline, then start over again editing the draft on device #3

Wh #1,
ed ng
th er
again editing the draft on device #3



Yes, you can use Gmail offline on desktop (via Chrome) to read, respond to, and search emails, which sync once connectivity returns. If you edit the same draft in two different offline sessions, **the last version to sync online "wins" and will overwrite the earlier version**. It is highly advised not to work on the same document or draft across multiple offline devices. Google Help +3

https://en.wikipedia.org/wiki/CAP_theorem



Sequential consistency...

Sequential consistency

(Read)R1. Alex arrives on Google Pixel 2 product page. He sees “Hurry up. Only 1 left!”

[**R2**] Bob arrives on mobile case product page. He sees “Hurry up. Only 1 left!”

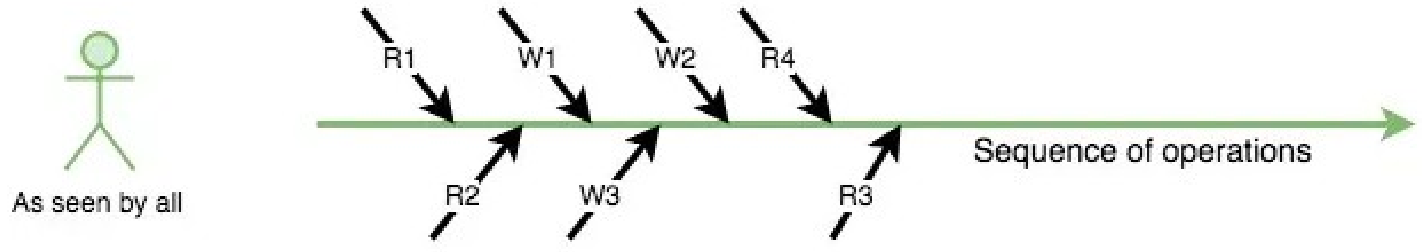
[**W1**] & [**W2**] Alex selects both the phone and the case and clicks on pay button. An update request is fired to first “block” the the mobile phone and then conditionally block the mobile case

[**W3**] Bob selects the case and clicks on pay button. An update request is fired to “block” the the mobile case

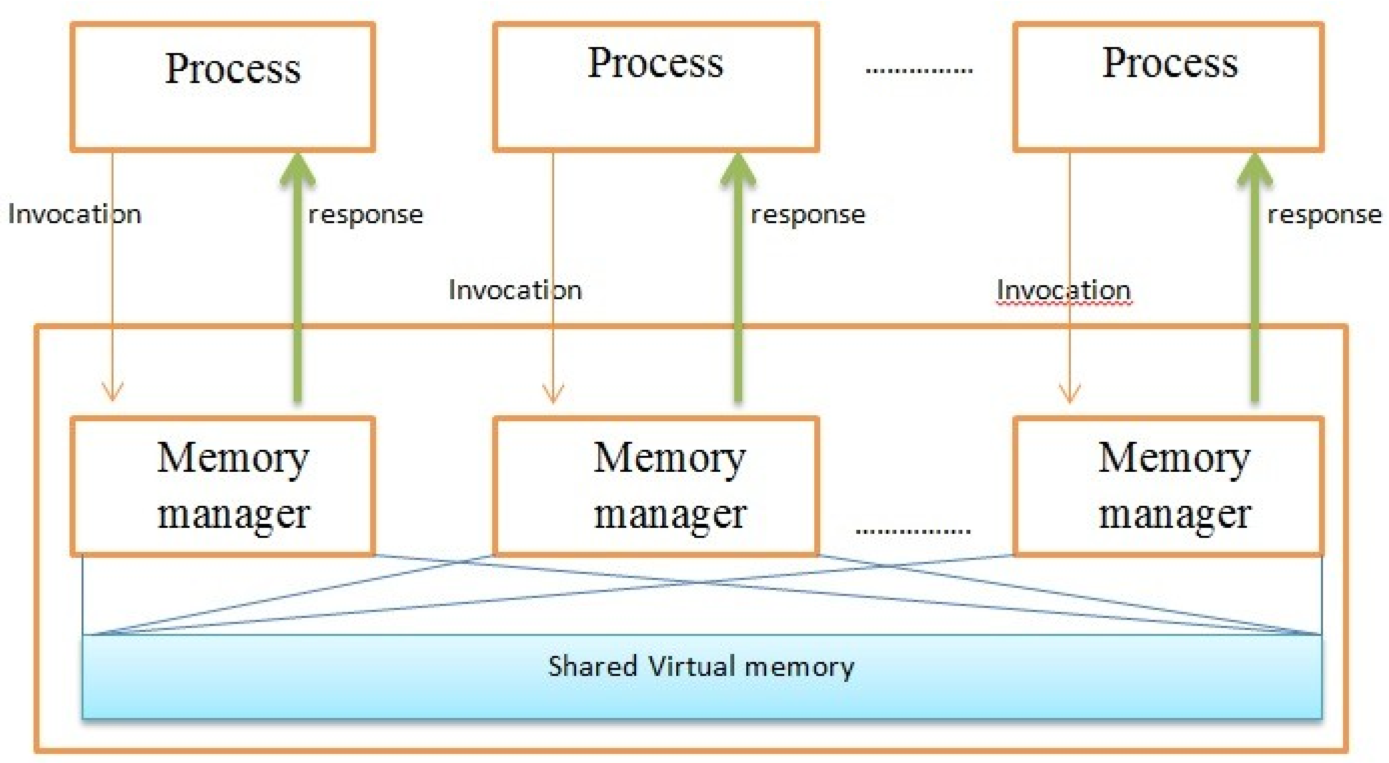
[**R3**] Bob’s blocking of mobile case is successful and he sees "Proceed to pay"

[**R4**] Alex’s blocking of mobile phone is successful but not the of mobile case. Alex sees “Sorry the combination is unavailable”.

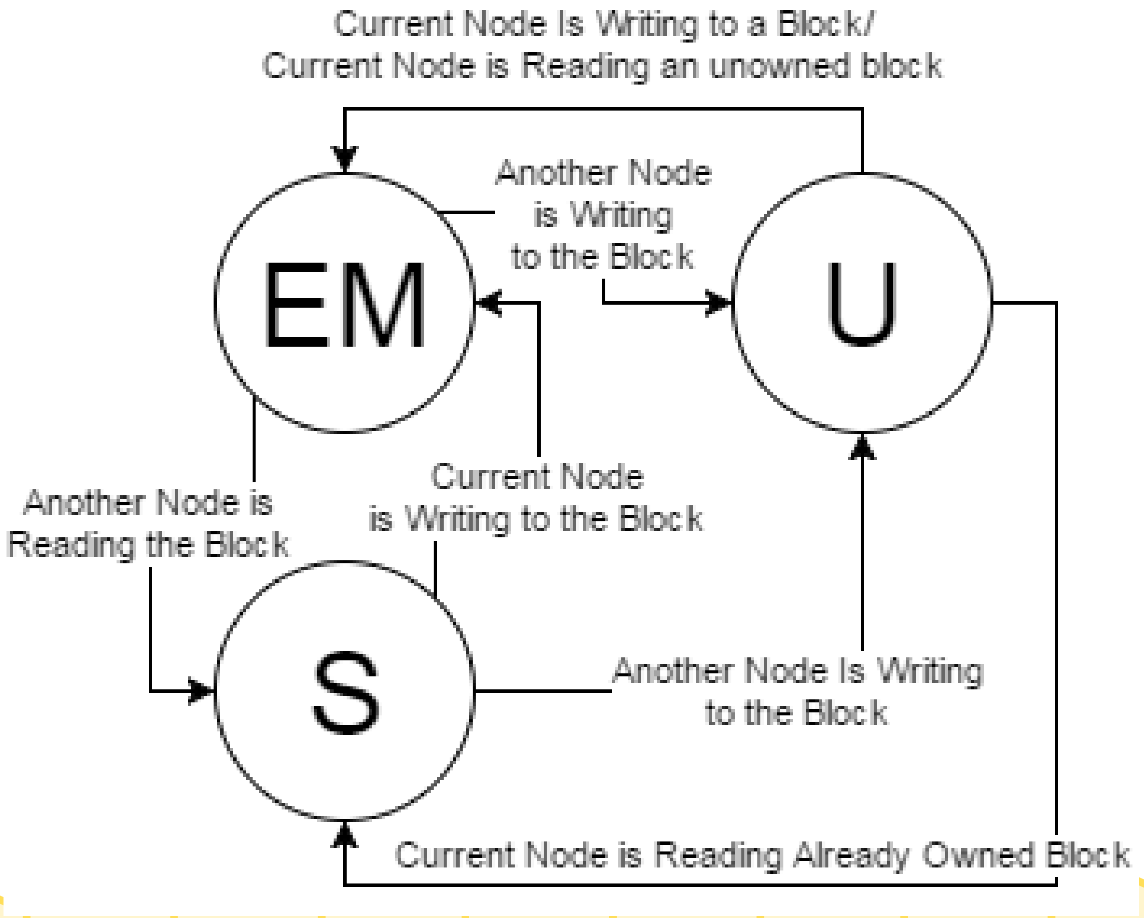
Sequential consistency



https://en.wikipedia.org/wiki/Distributed_shared_memory



Distributed shared memory

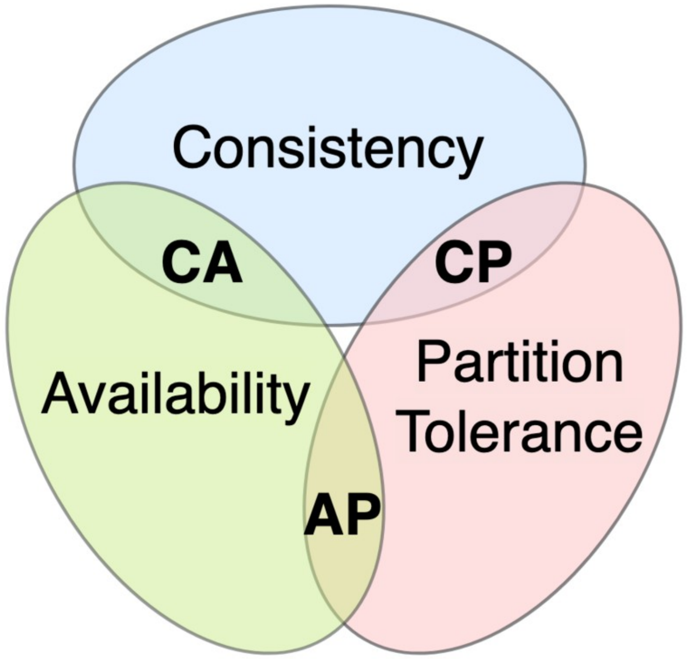

Current Node Is Writing to a Block/
Current Node is Reading an unowned block
EM
Another Node
is Writing
to the Block
U
Another Node is
Reading the Block
Current Node
is Writing to the Block
S
Another Node Is Writing
to the Block
Current Node is Reading Already Owned Block

What if a node fails to respond?

What if a node fails to respond?

A block of memory can simply become unavailable, because the tradeoff is towards consistency and partition tolerance.

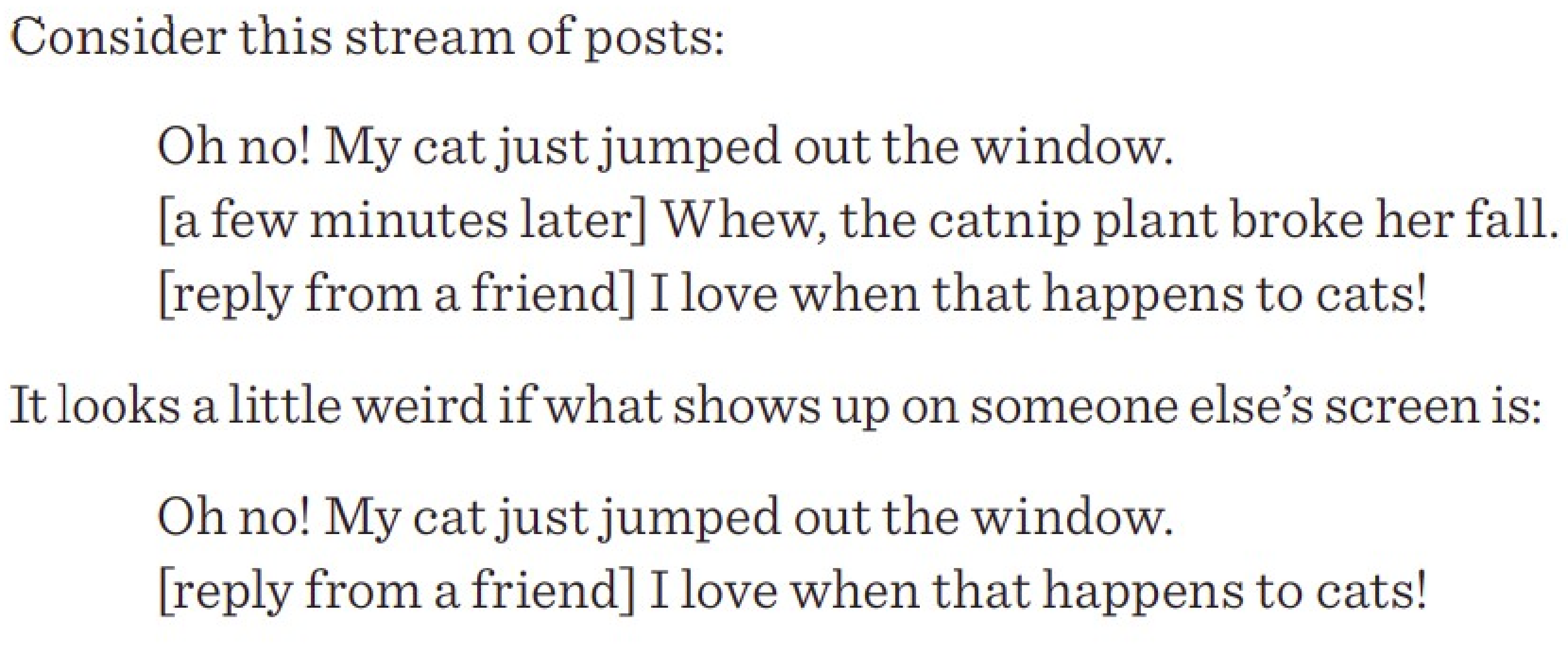
https://en.wikipedia.org/wiki/CAP_theorem



What if we really care about availability and we want partition tolerance? We have to relax consistency.

Causal consistency

https://www.usenix.org/system/files/login/articles/08_lloyd_41-43_online.pdf



Consider this stream of posts:

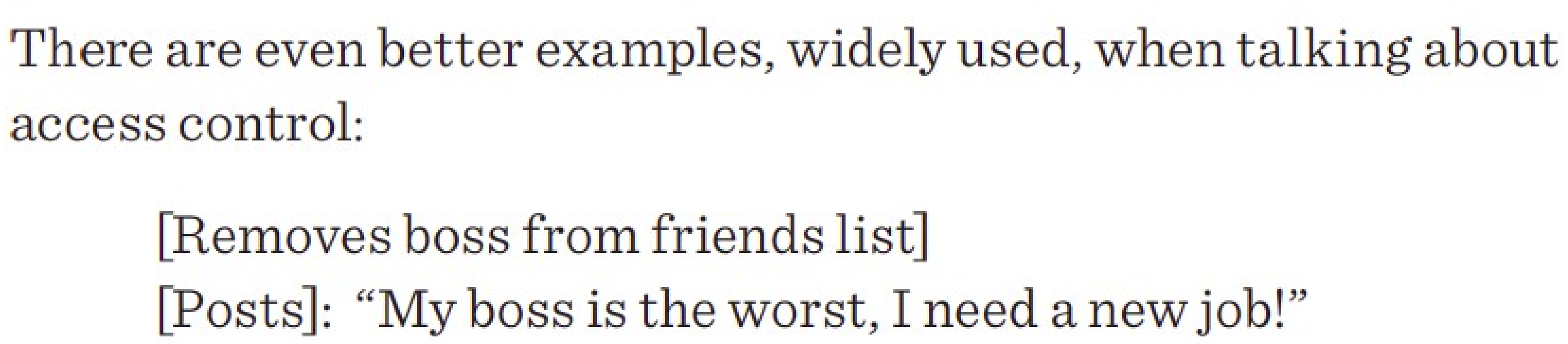
Oh no! My cat just jumped out the window.
[a few minutes later] Whew, the catnip plant broke her fall.
[reply from a friend] I love when that happens to cats!

It looks a little weird if what shows up on someone else's screen is:

Oh no! My cat just jumped out the window.
[reply from a friend] I love when that happens to cats!

Causal consistency

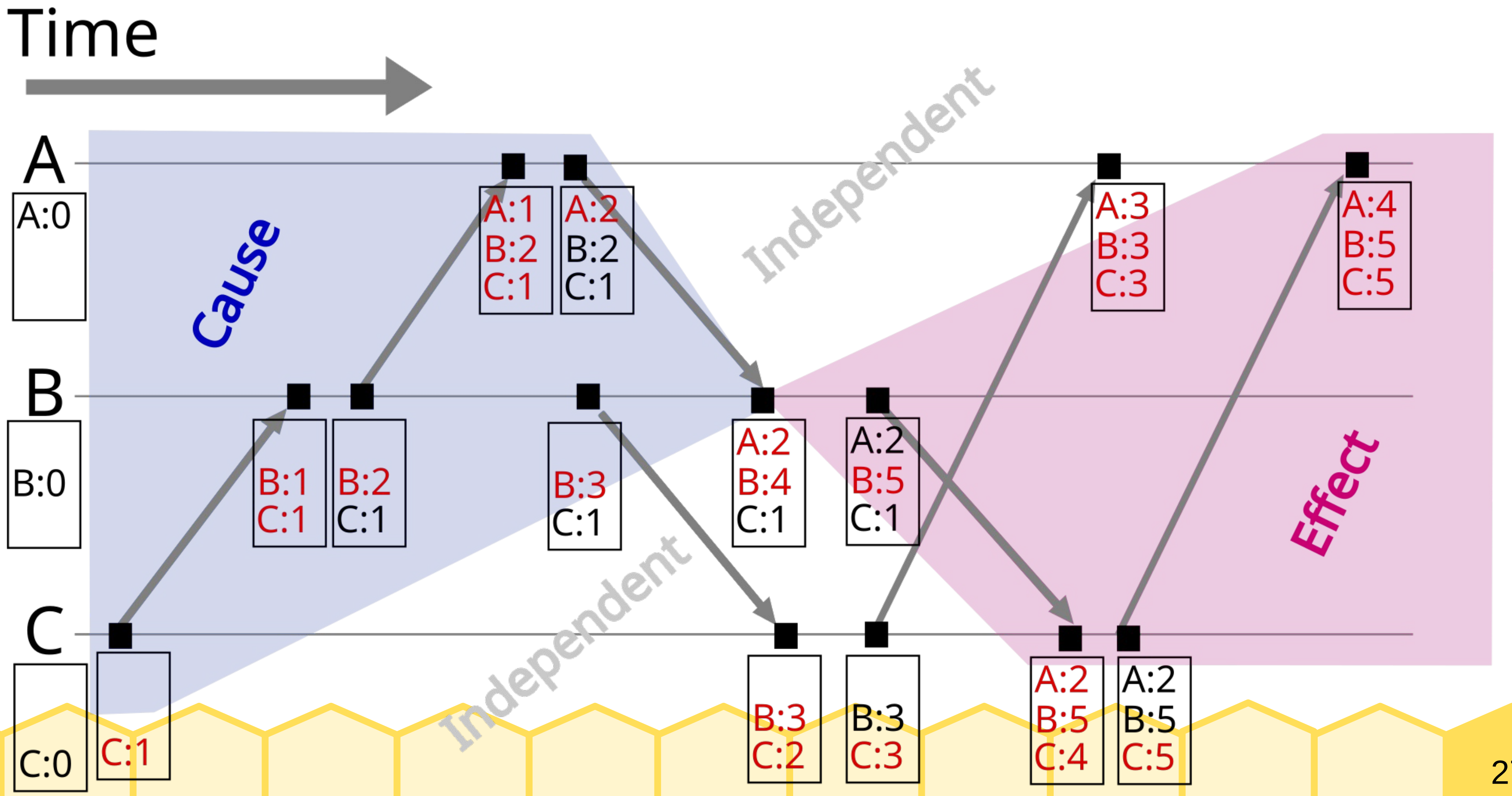
https://www.usenix.org/system/files/login/articles/08_lloyd_41-43_online.pdf



There are even better examples, widely used, when talking about access control:

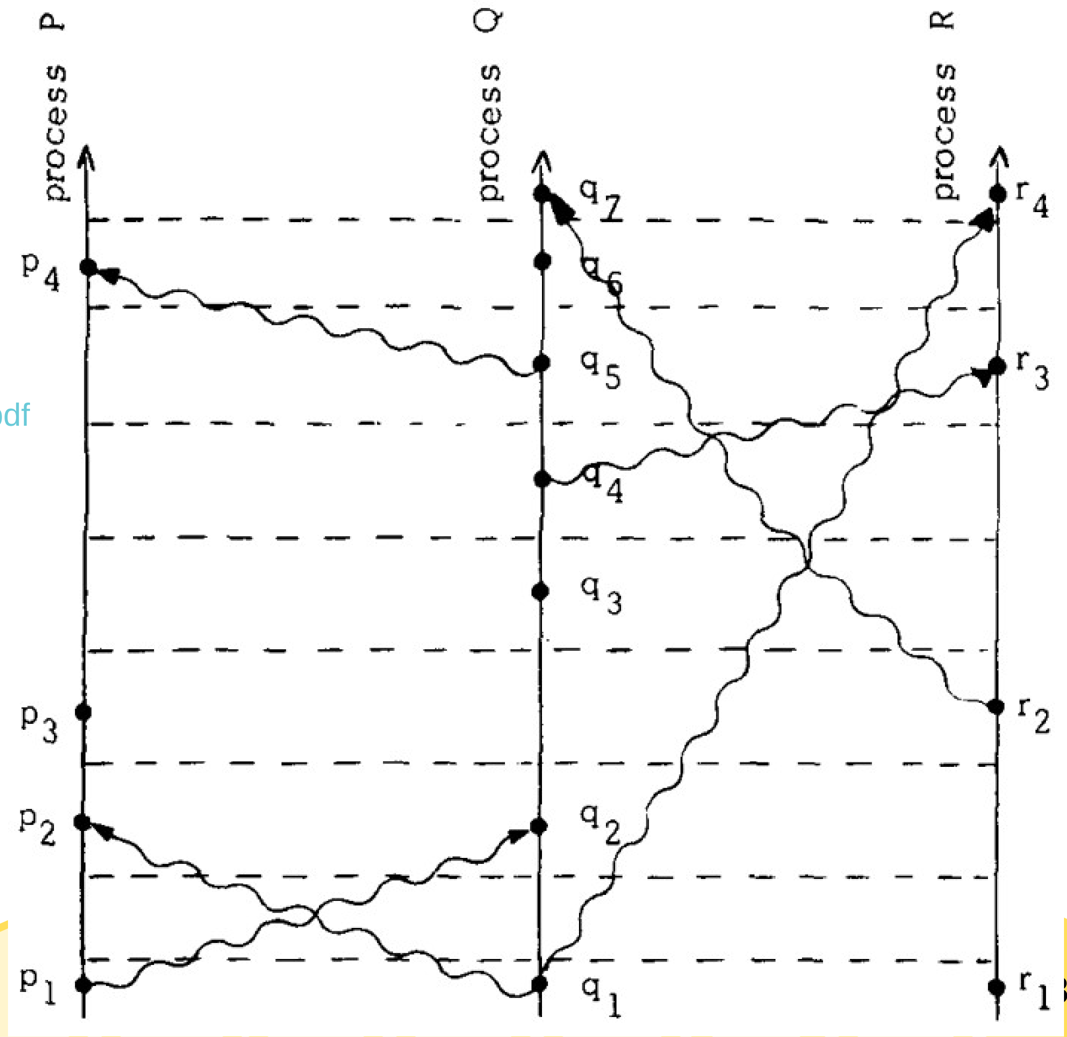
[Removes boss from friends list]
[Posts]: "My boss is the worst, I need a new job!"

https://en.wikipedia.org/wiki/Vector_clock



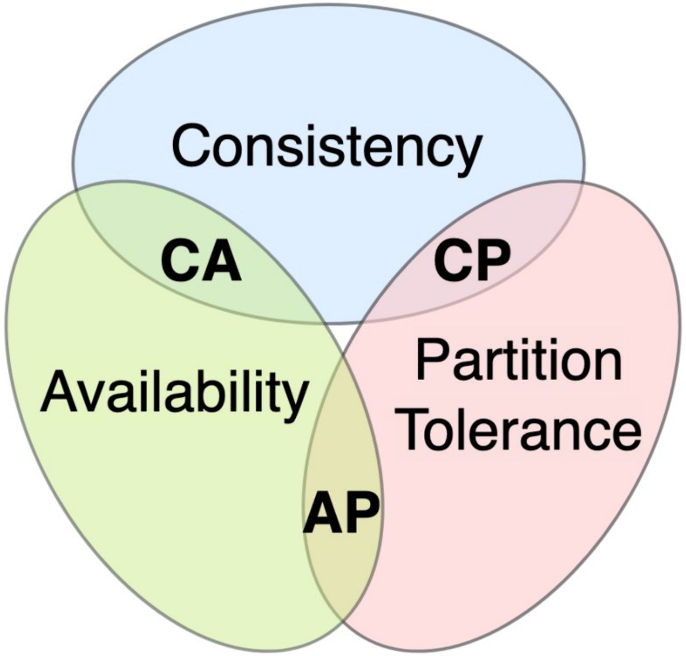
https://lamport.azurewebsites.net/pubs/time-clocks.pdf

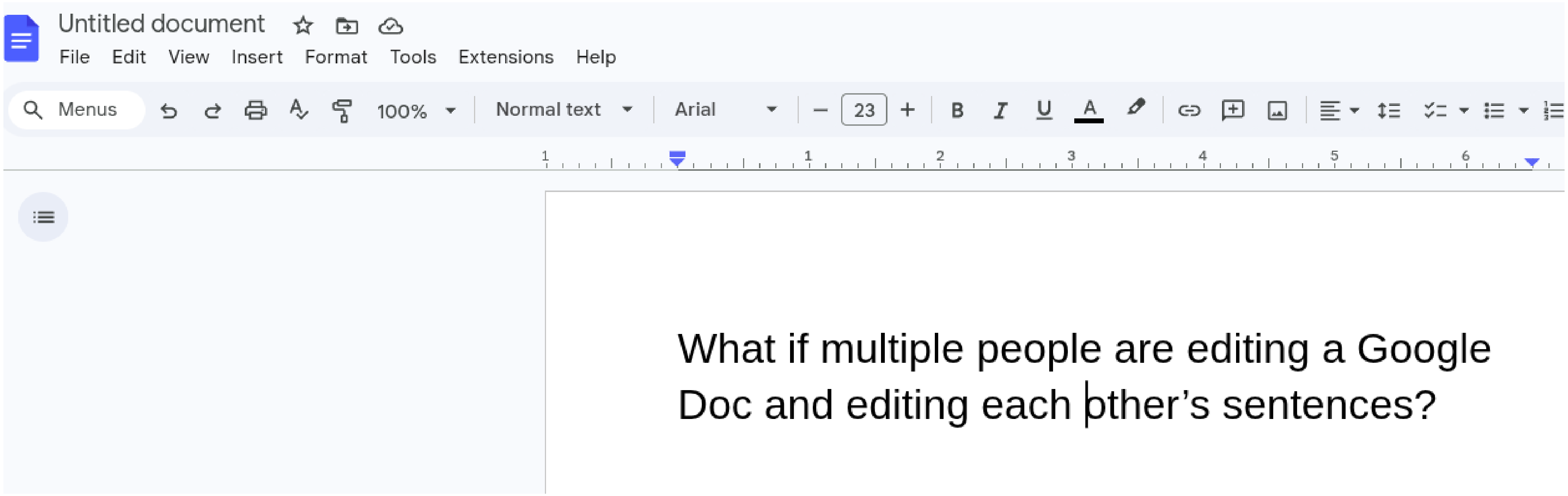
Fig. 3.



With causal consistency we can have <u>both</u> availability and partition tolerance.

https://en.wikipedia.org/wiki/CAP_theorem





What if multiple people are editing a Google Doc and editing each other's sentences?

We also want eventual consistency.
(Need some kind of conflict resolution).

Consistency models

- Strict consistency
  - All reads and writes in the same order for every process
- Sequential consistency
  - Reads and writes for a process in order, all writes in FIFO order
- Causal consistency
  - Potentially causal writes seen in same order
  - Concurrent writes can be in a different order
- PRAM, many others...

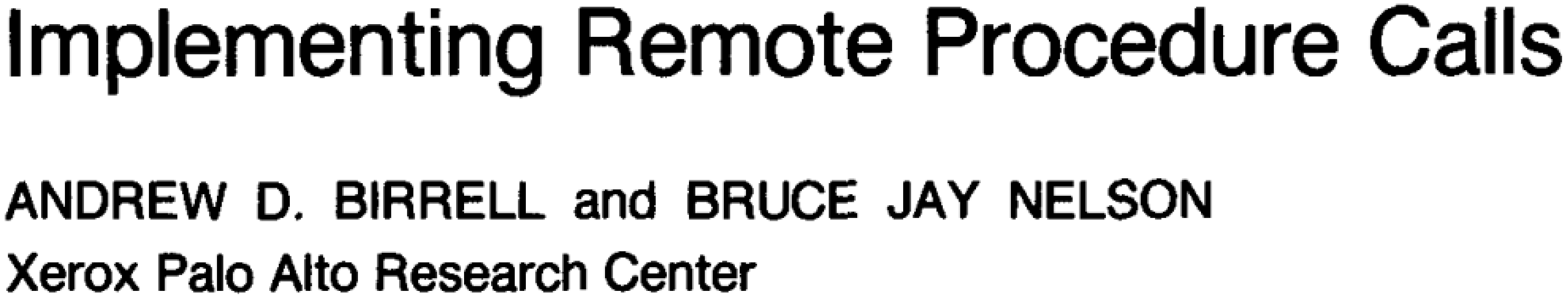
But, let’s step back and look at a more simple form of distributed computing…

RPC (Remote Procedure Calls)

Remote Procedure Calls

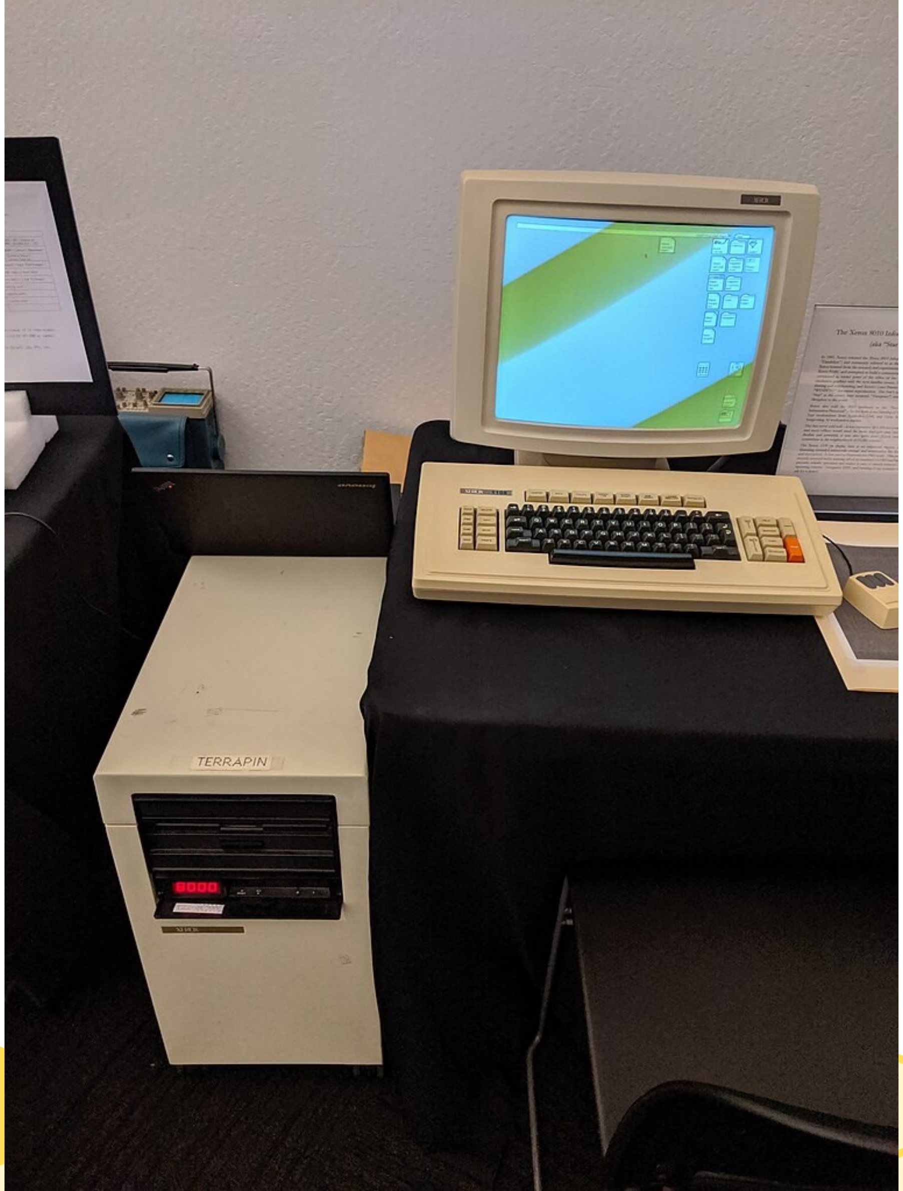
- Basic idea: use something programmers are already familiar with (calling a procedure and it returning a value)
  - Make distributed computation easy
  - Not rocket science
    - At least not on the surface
  - Heavily used in practice
  - Caller or callee can crash, doesn’t break everything
- https://jedcrandall.github.io/courses/cse536spring2024/birrell842.pdf

https://jedcrandall.github.io/courses/cse536spring2024/birrell842.pdf

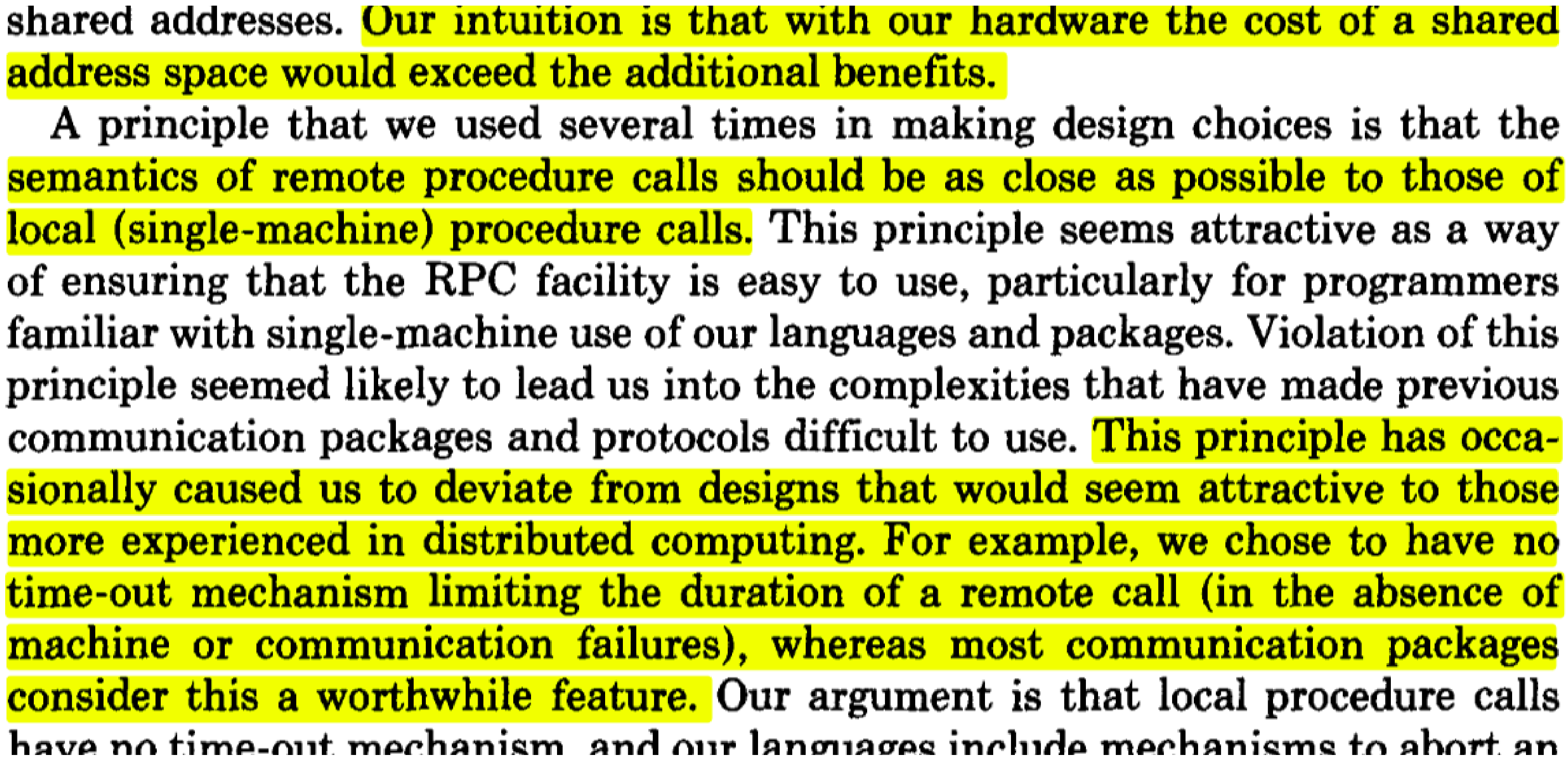


Implementing Remote Procedure Calls

ANDREW D. BIRRELL and BRUCE JAY NELSON
Xerox Palo Alto Research Center



https://en.m.wikipedia.org/wiki/Xerox_Star



shared addresses. Our intuition is that with our hardware the cost of a shared address space would exceed the additional benefits.

A principle that we used several times in making design choices is that the semantics of remote procedure calls should be as close as possible to those of local (single-machine) procedure calls. This principle seems attractive as a way of ensuring that the RPC facility is easy to use, particularly for programmers familiar with single-machine use of our languages and packages. Violation of this principle seemed likely to lead us into the complexities that have made previous communication packages and protocols difficult to use. This principle has occasionally caused us to deviate from designs that would seem attractive to those more experienced in distributed computing. For example, we chose to have no time-out mechanism limiting the duration of a remote call (in the absence of machine or communication failures), whereas most communication packages consider this a worthwhile feature. Our argument is that local procedure calls have no time-out mechanism, and our languages include mechanisms to abort an

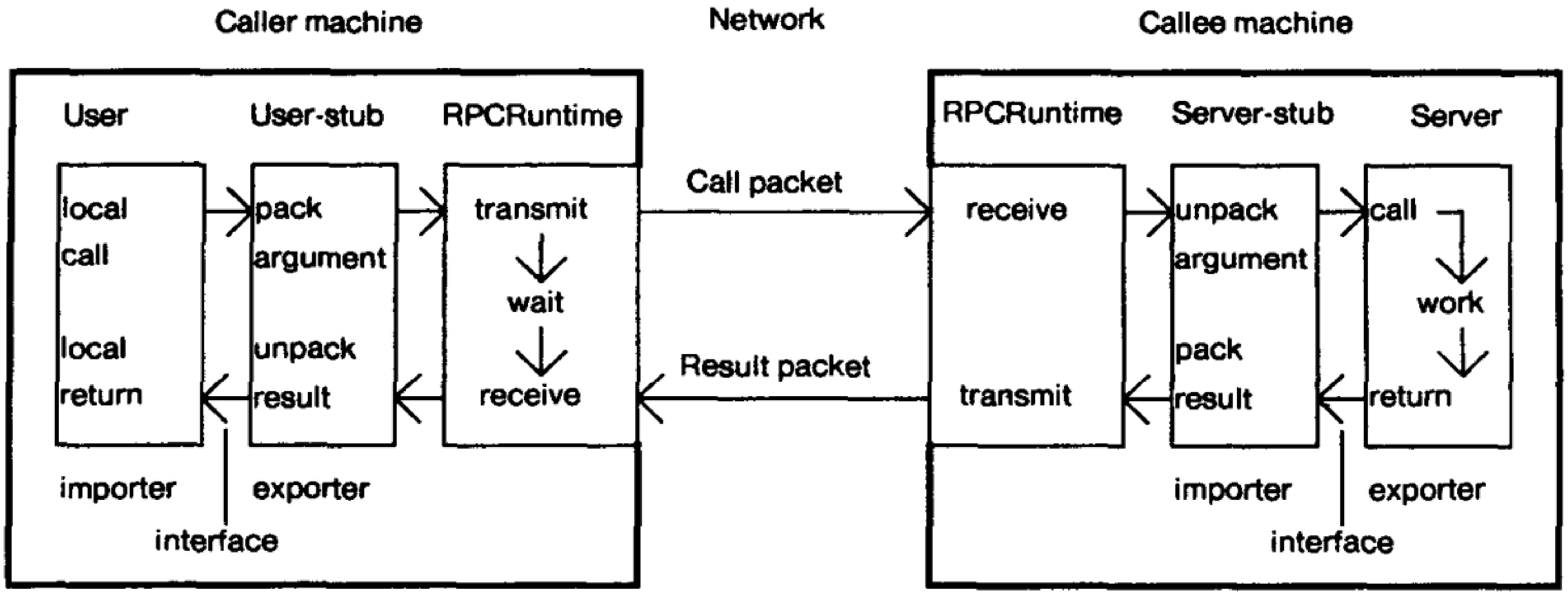
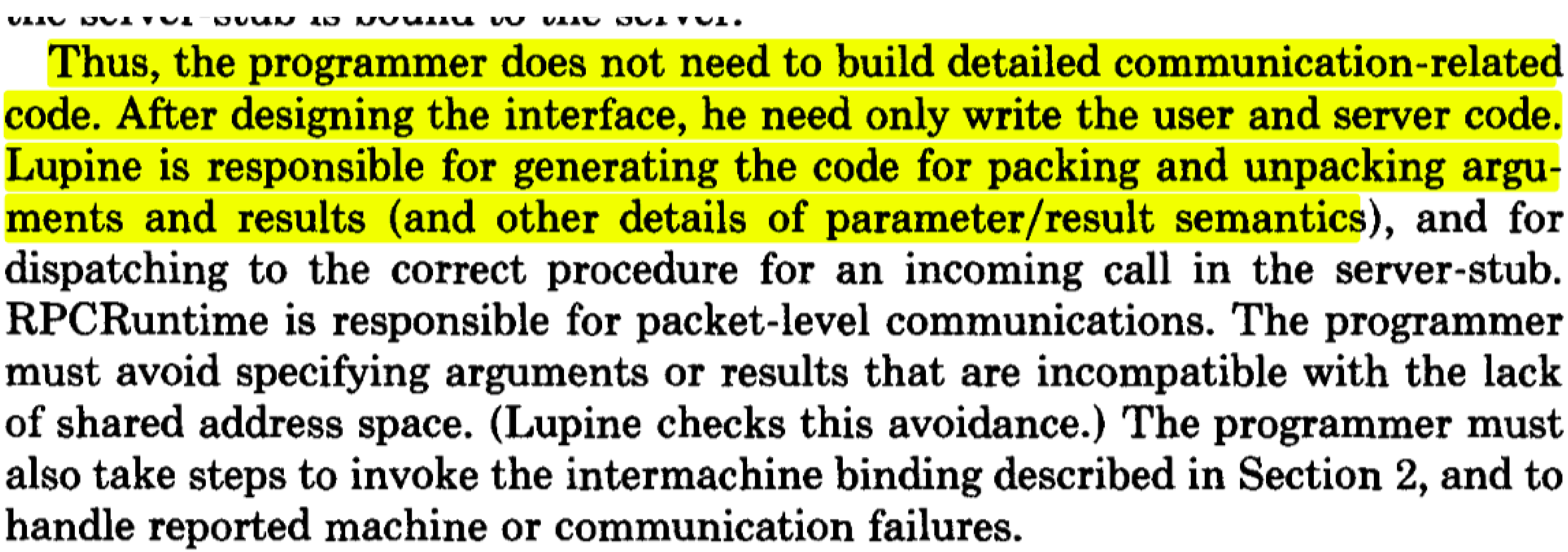


Fig. 1. The components of the system, and their interactions for a simple call.

Terminology

- Marshalling (not in the paper, but implied).. packing and unpacking (unmarshalling) the parameters
  - Necessary because of differences in machines, representations
  - Think XML, JSON, protobuf, ASN.1, ...



the server-stub is bound to the server.

Thus, the programmer does not need to build detailed communication-related code. After designing the interface, he need only write the user and server code. Lupine is responsible for generating the code for packing and unpacking arguments and results (and other details of parameter/result semantics), and for dispatching to the correct procedure for an incoming call in the server-stub. RPCRuntime is responsible for packet-level communications. The programmer must avoid specifying arguments or results that are incompatible with the lack of shared address space. (Lupine checks this avoidance.) The programmer must also take steps to invoke the intermachine binding described in Section 2, and to handle reported machine or communication failures.

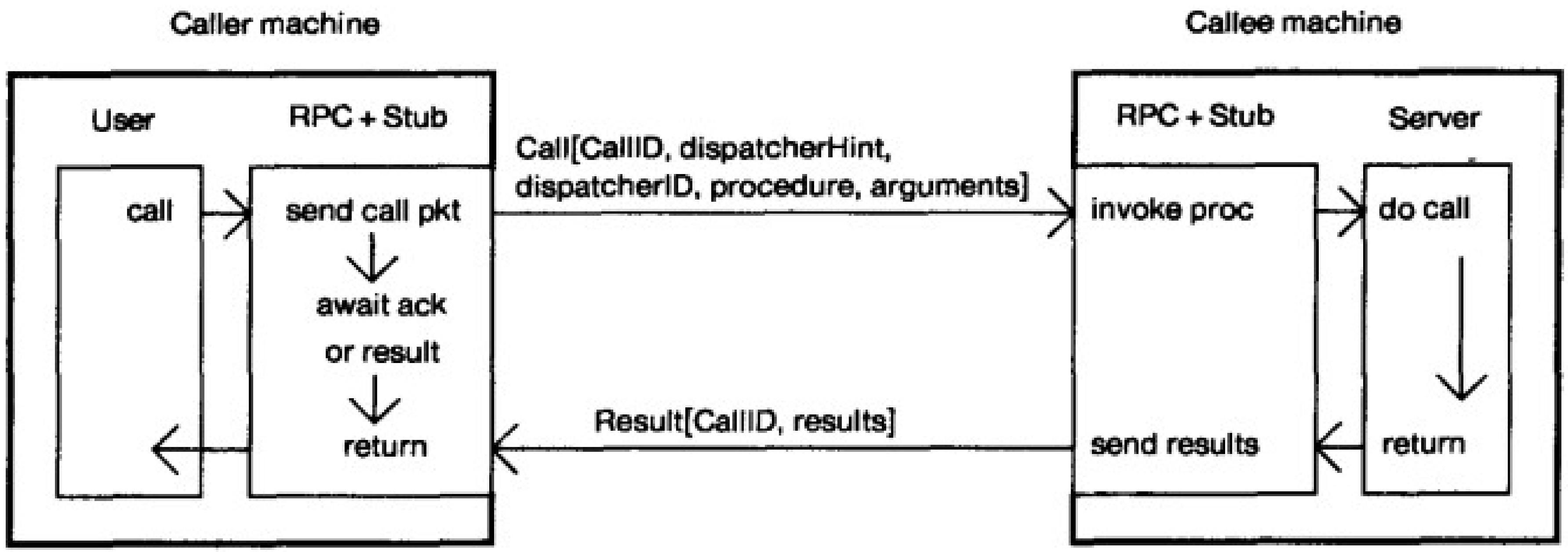


Fig. 3. The packets transmitted during a simple call.

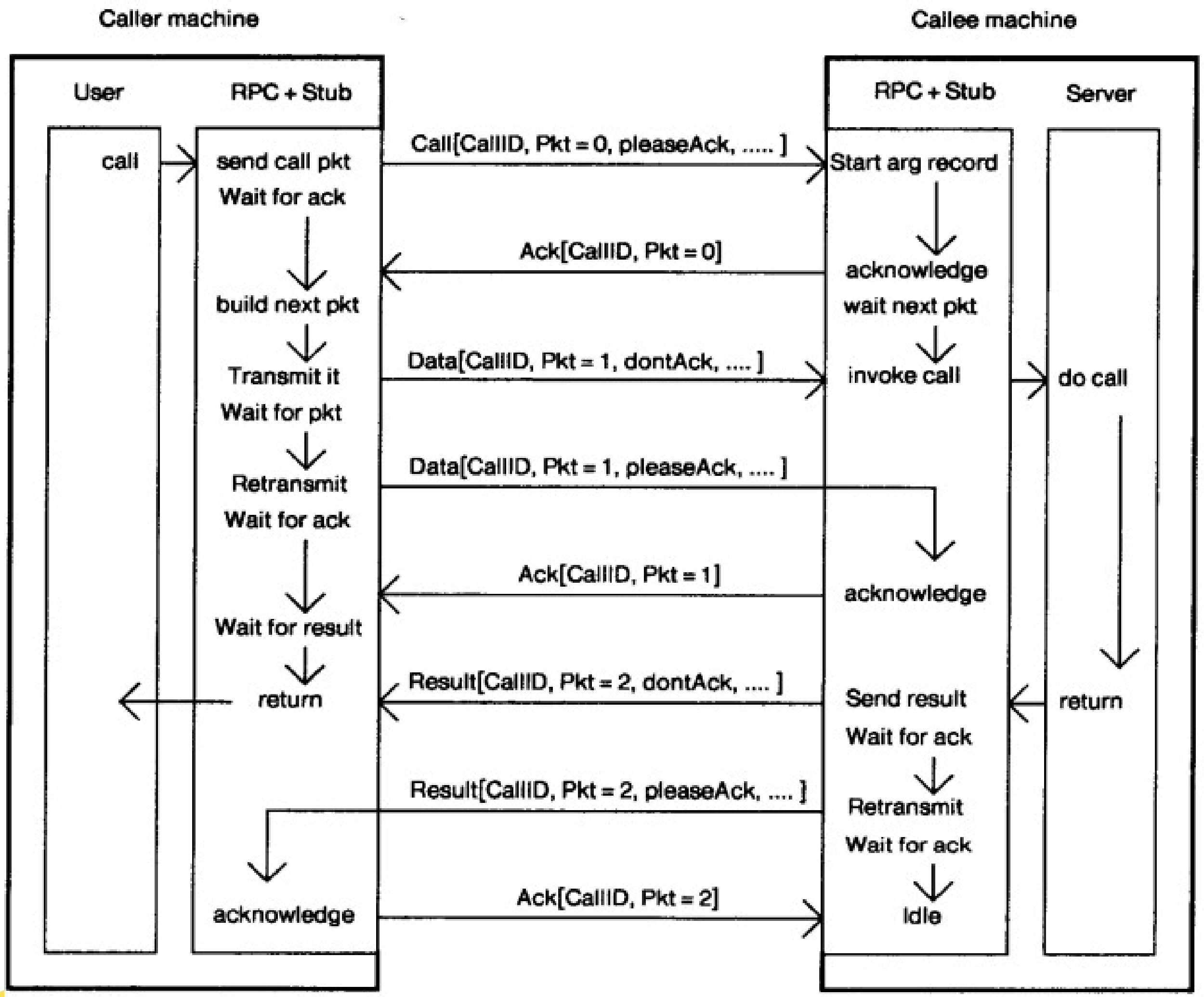
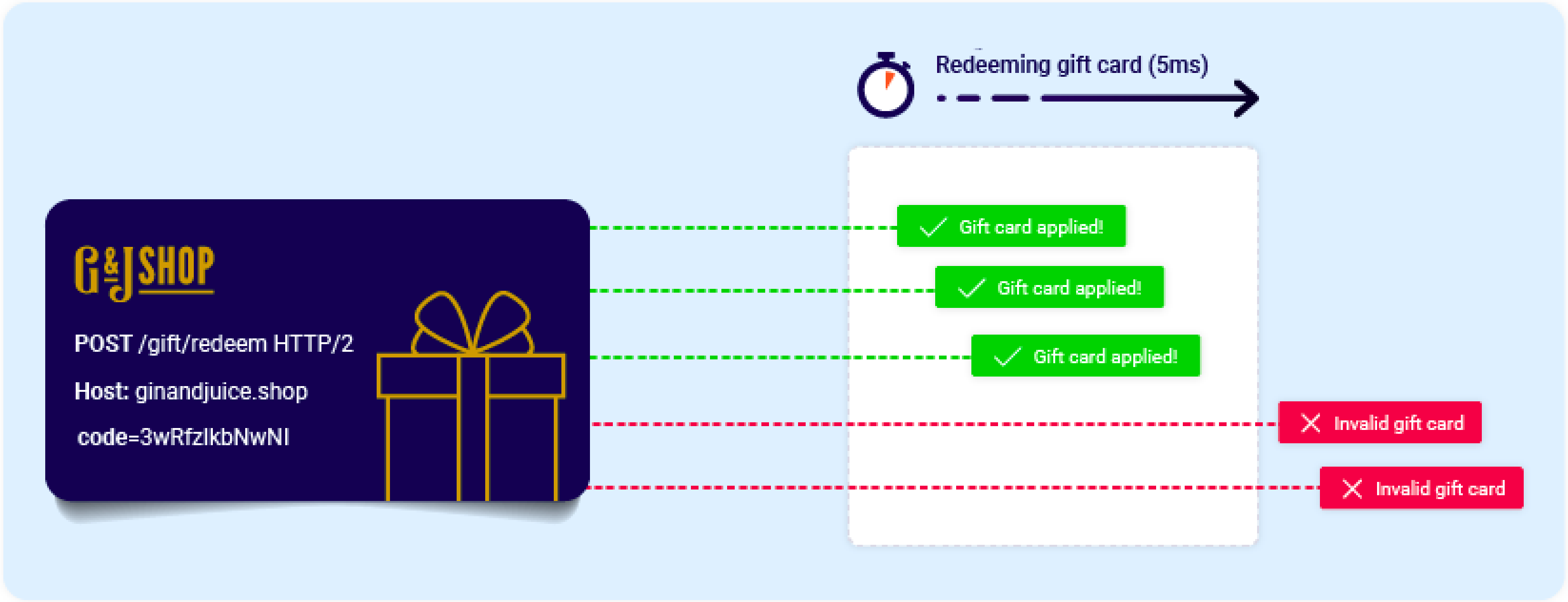
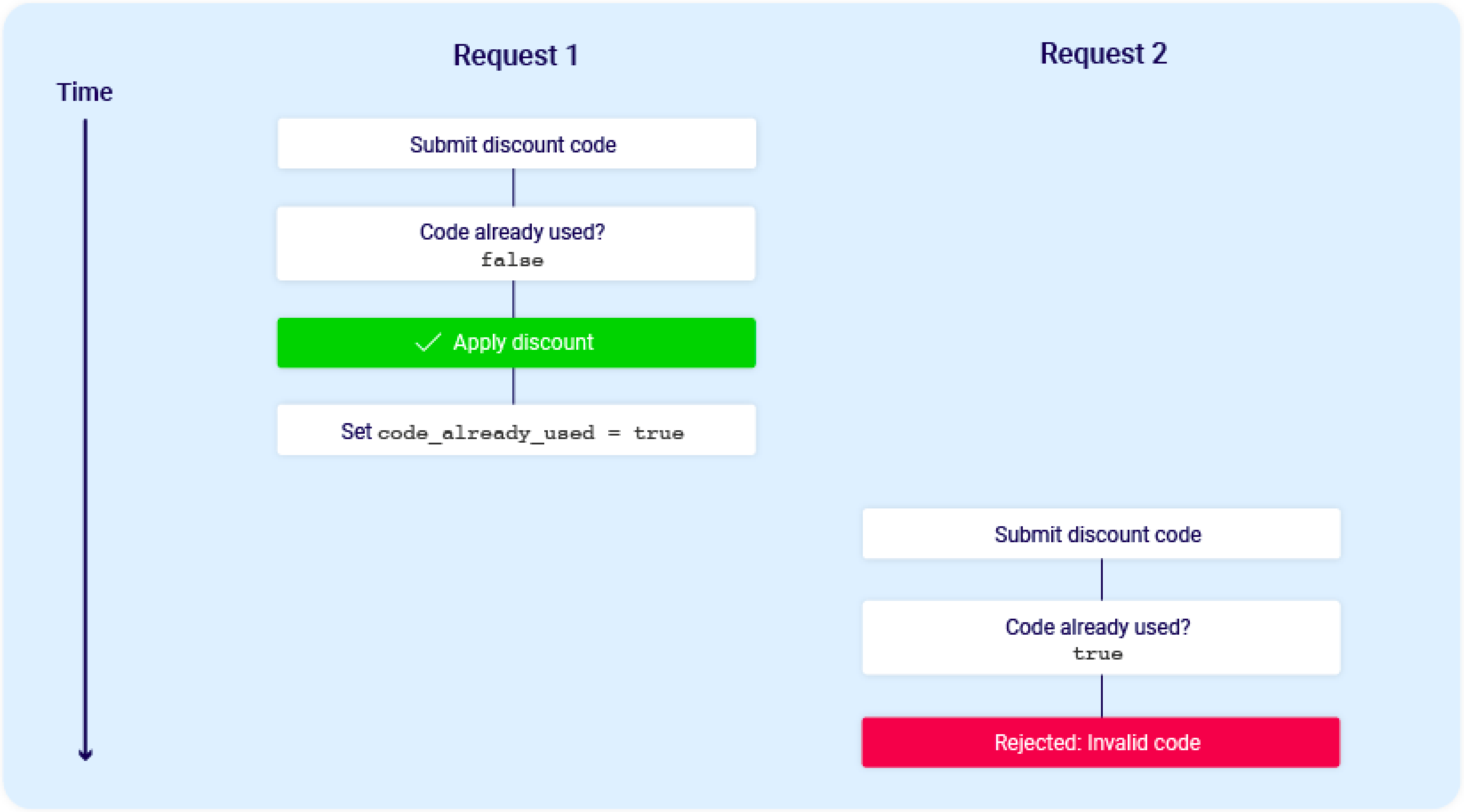


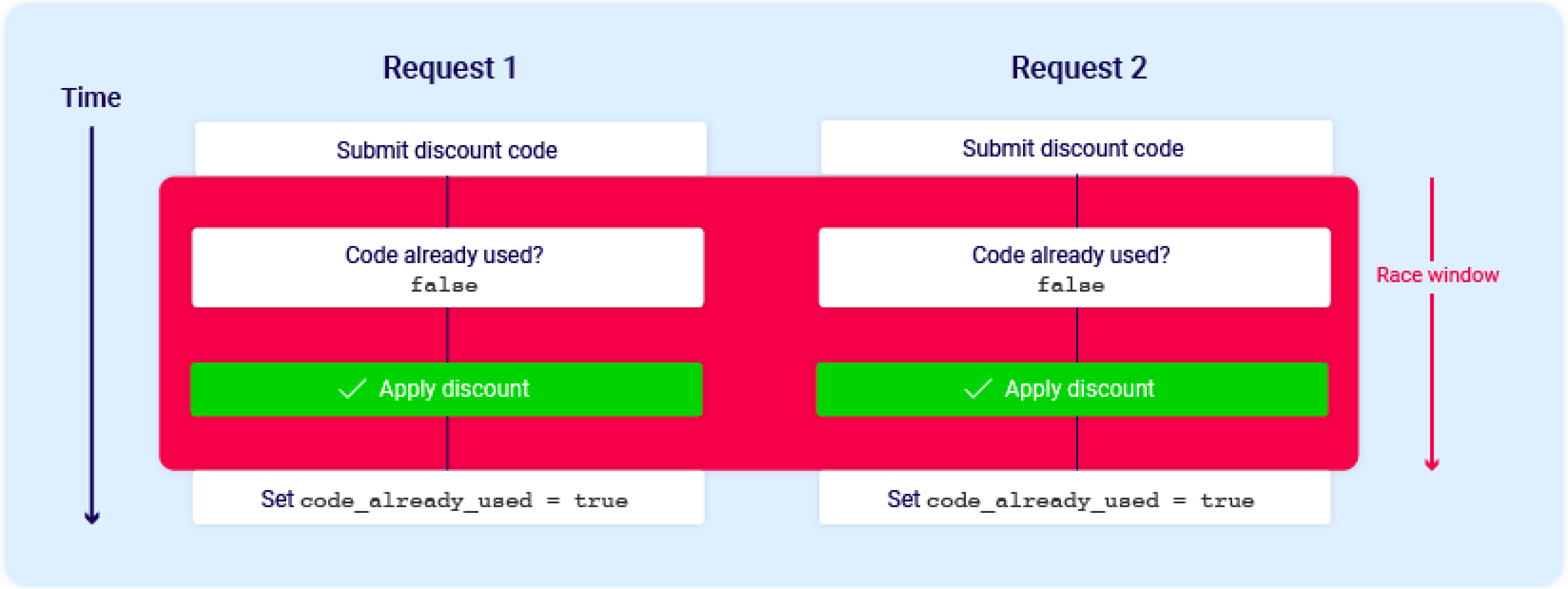
Fig. 4. A complicated call. The arguments occupy two packets. The call duration is long enough to require retransmission of the last argument packet requesting an acknowledgment, and the result packet is retransmitted requesting an acknowledgment because no subsequent call arrived.

Where does the “this solves our concurrency and race conditions problem” part come?

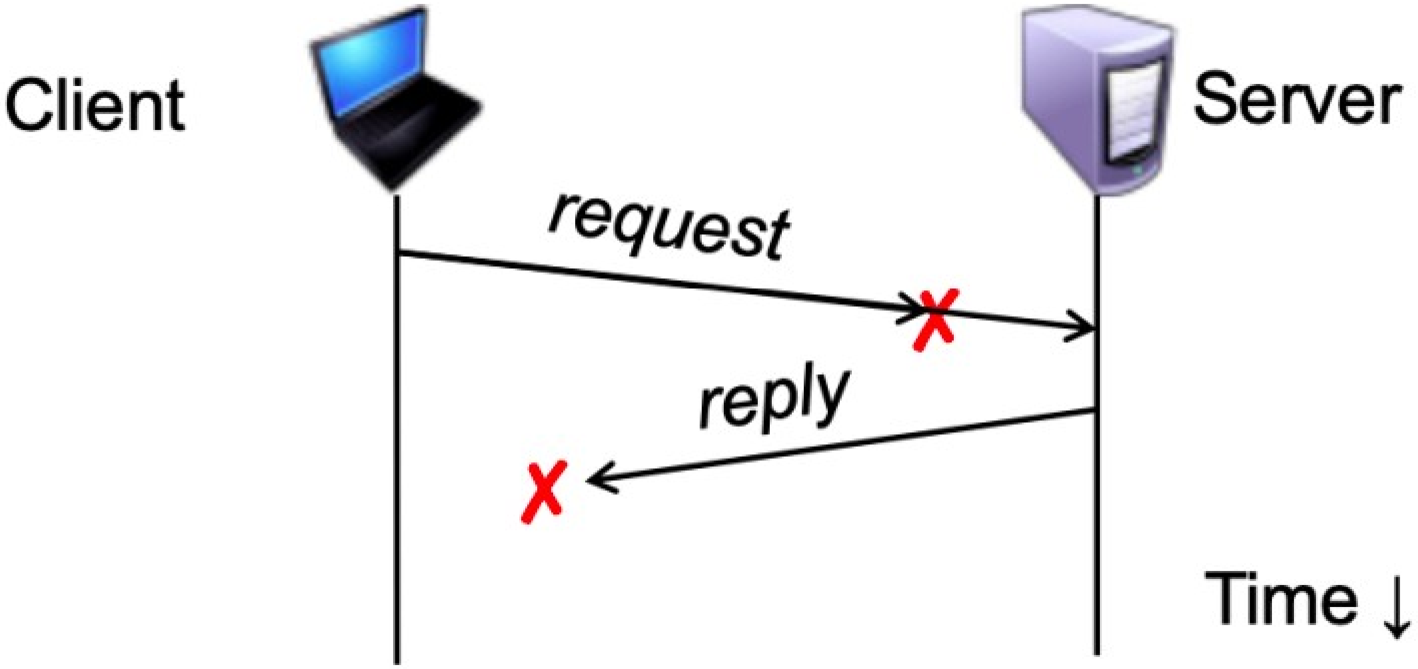


```
POST /gift/redeem HTTP/2
Host: ginandjuice.shop
code=3wRfzlkbNwNI
```


Request 1
Request 2
Time
Submit discount code
Code already used?
false
✓ Apply discount
Set code_already_used = true
Submit discount code
Code already used?
true
Rejected: Invalid code


Request 1
Request 2
Time
Submit discount code
Submit discount code
Code already used?
false
Code already used?
false
Race window
✓ Apply discount
✓ Apply discount
Set code_already_used = true
Set code_already_used = true

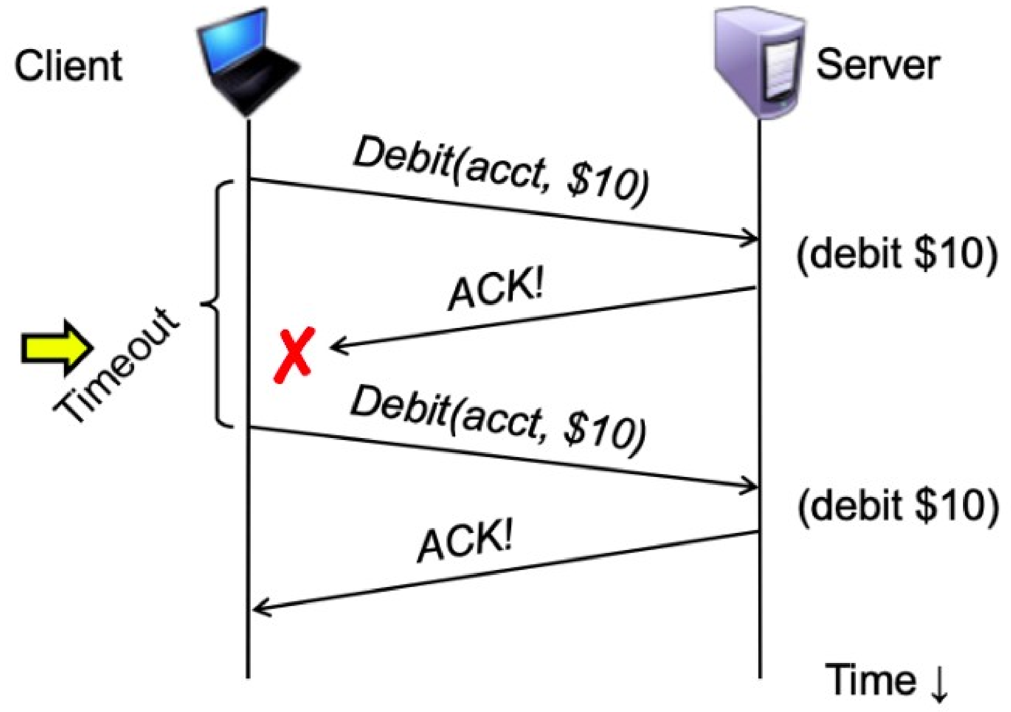
https://changlousys.github.io/CS4740/spring24/slides/Course5-RPC.pdf



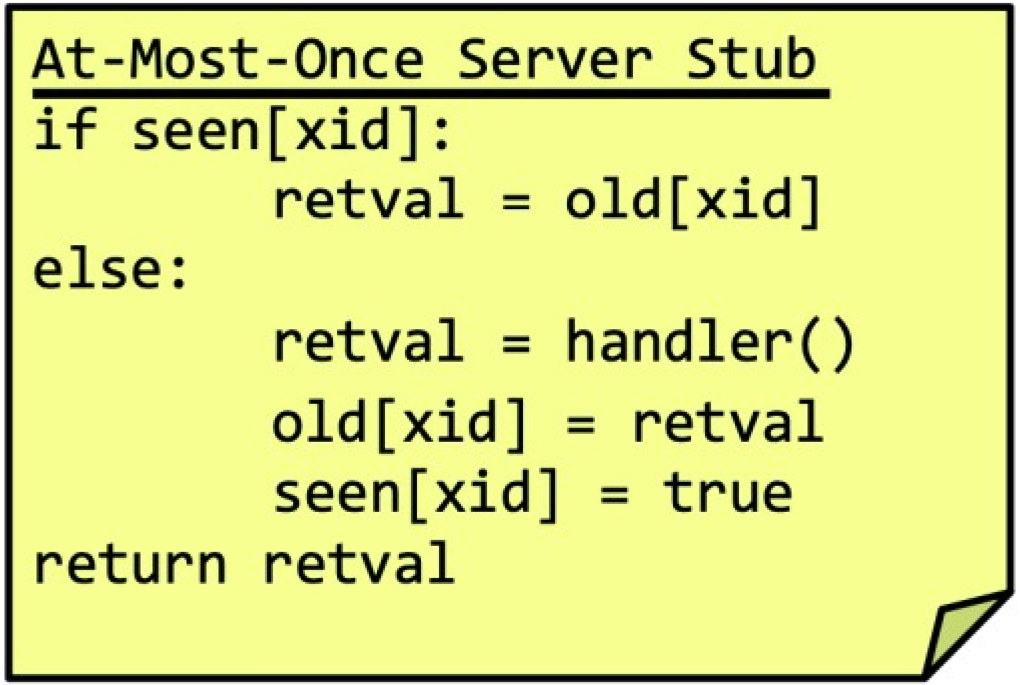
RPC semantics

- At least once
- At most once
- Exactly once

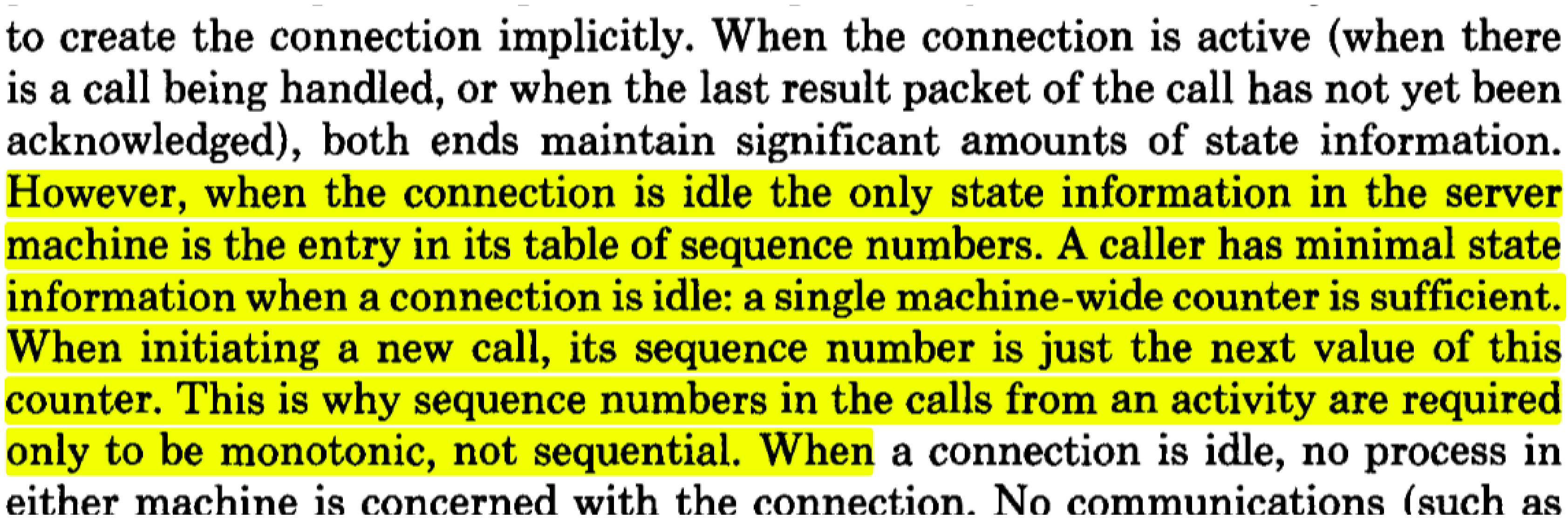
At least once



At most once



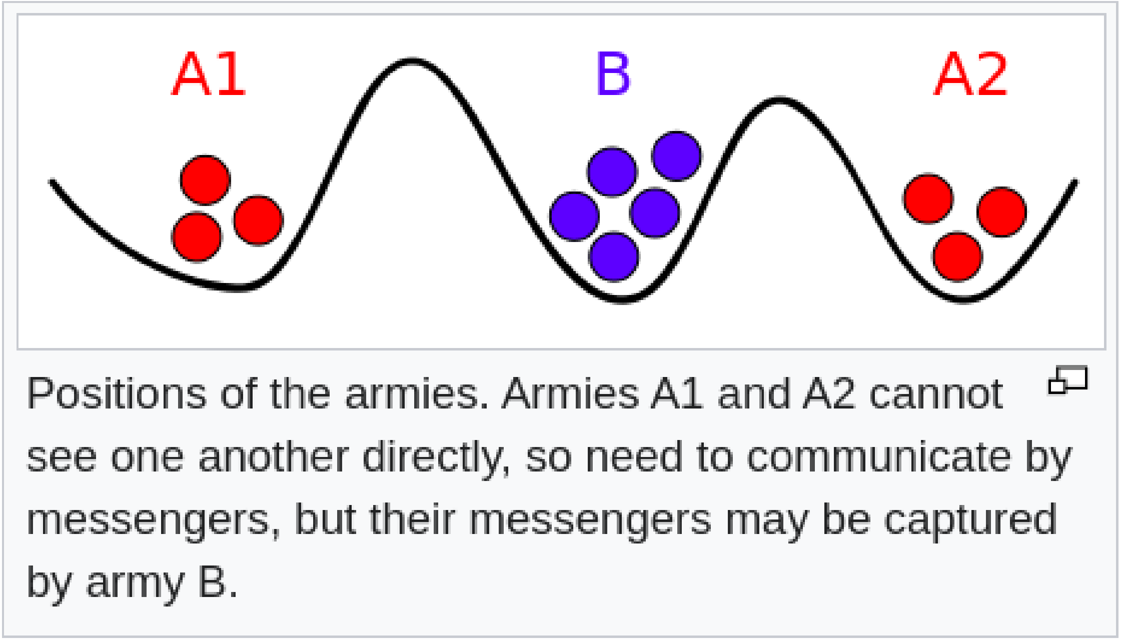
```
At-Most-Once Server Stub
if seen[xid]:
        retval = old[xid]
else:
        retval = handler()
        old[xid] = retval
        seen[xid] = true
return retval
```


to create the connection implicitly. When the connection is active (when there is a call being handled, or when the last result packet of the call has not yet been acknowledged), both ends maintain significant amounts of state information. However, when the connection is idle the only state information in the server machine is the entry in its table of sequence numbers. A caller has minimal state information when a connection is idle: a single machine-wide counter is sufficient. When initiating a new call, its sequence number is just the next value of this counter. This is why sequence numbers in the calls from an activity are required only to be monotonic, not sequential. When a connection is idle, no process in either machine is concerned with the connection. No communications (such as

What about exactly once?

Two Generals' Problem

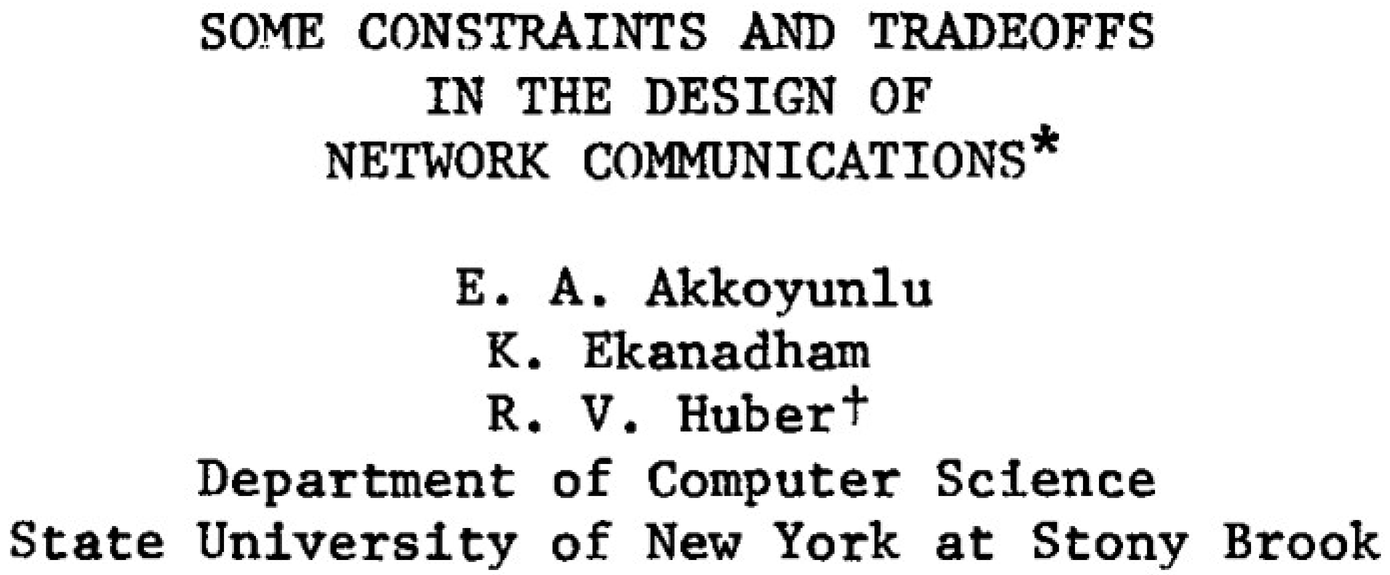
- https://en.wikipedia.org/wiki/Two_Generals%27_Problem



Positions of the armies. Armies A1 and A2 cannot see one another directly, so need to communicate by messengers, but their messengers may be captured by army B.

Two Generals' Problem

- A1 and A2 need to agree on when to attack
  - Same as A and B in our example unambiguously knowing the state of their own connection (open, half-open, closed?), and other state about the connection

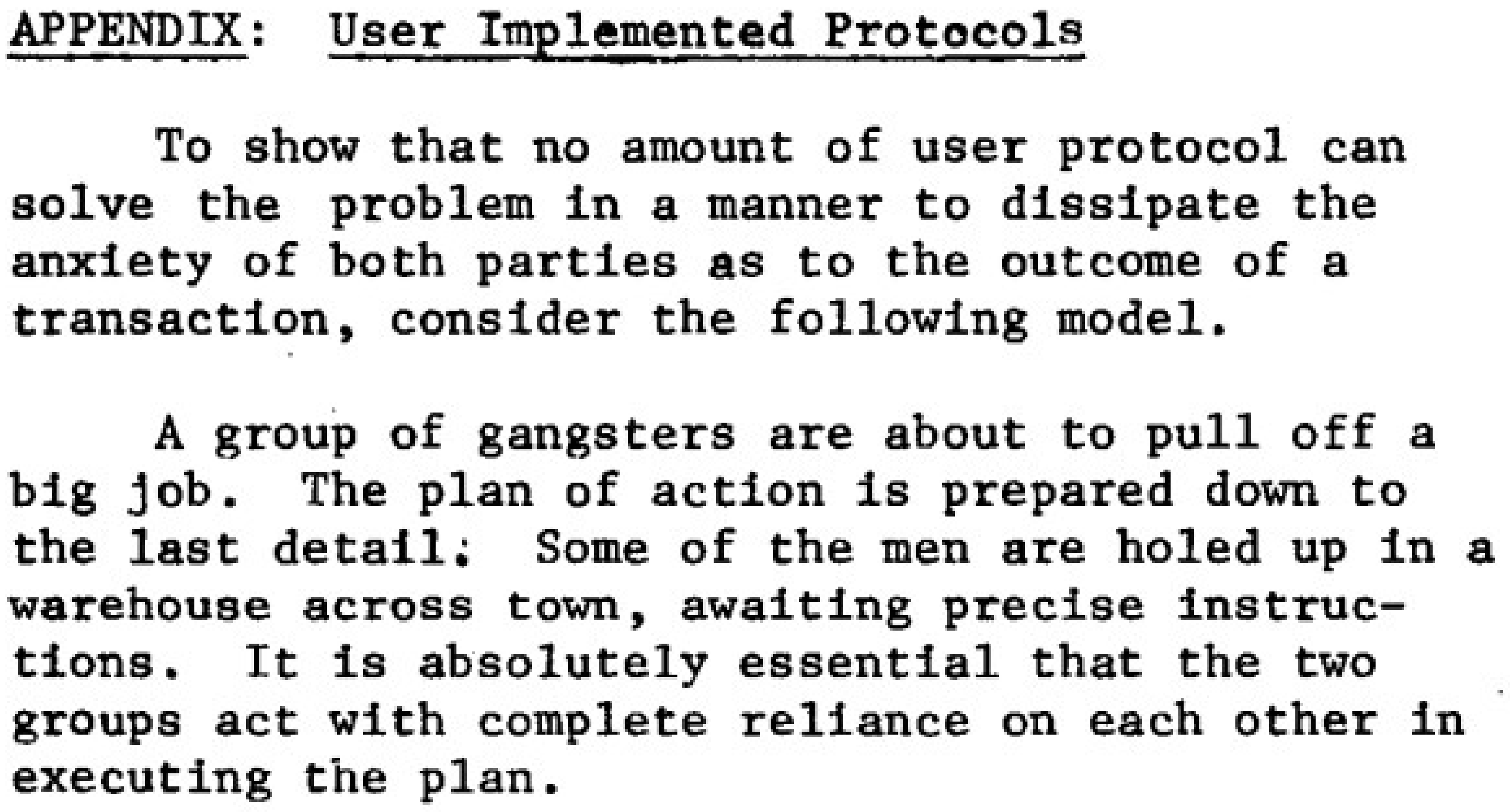


SOME CONSTRAINTS AND TRADEOFFS
IN THE DESIGN OF
NETWORK COMMUNICATIONS*

E. A. Akkoyunlu
K. Ekanadham
R. V. Huber†
Department of Computer Science
State University of New York at Stony Brook

SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles • November 1975 • Pages 67–74

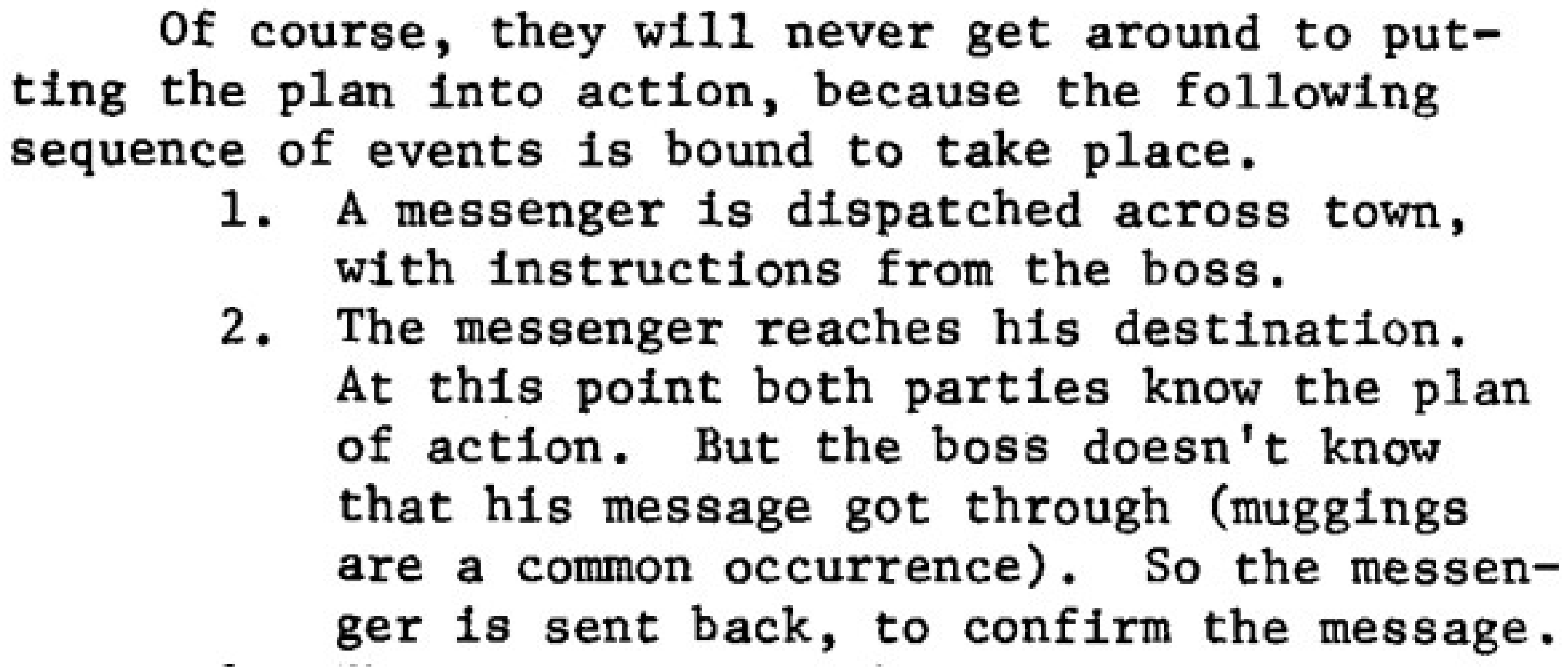
• https://doi.org/10.1145/800213.806523



APPENDIX: User Implemented Protocols

To show that no amount of user protocol can solve the problem in a manner to dissipate the anxiety of both parties as to the outcome of a transaction, consider the following model.

A group of gangsters are about to pull off a big job. The plan of action is prepared down to the last detail: Some of the men are holed up in a warehouse across town, awaiting precise instructions. It is absolutely essential that the two groups act with complete reliance on each other in executing the plan.



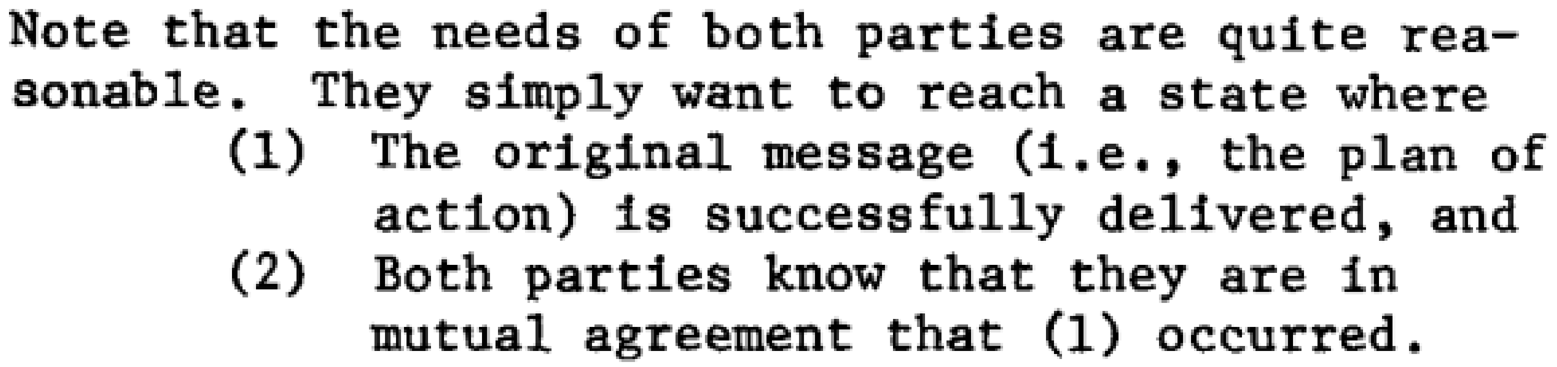
Of course, they will never get around to putting the plan into action, because the following sequence of events is bound to take place.

1. A messenger is dispatched across town, with instructions from the boss.
2. The messenger reaches his destination. At this point both parties know the plan of action. But the boss doesn't know that his message got through (muggings are a common occurrence). So the messenger is sent back, to confirm the message.

3. The messenger reaches the boss safely. Now, everybody knows the message got through. Of course, the men in the warehouse are not aware that step 3 occurred, and must be reassured. Off goes the messenger.
4. Now the men in the warehouse too know that step 3 was successful, but unless they communicate their awareness...

....

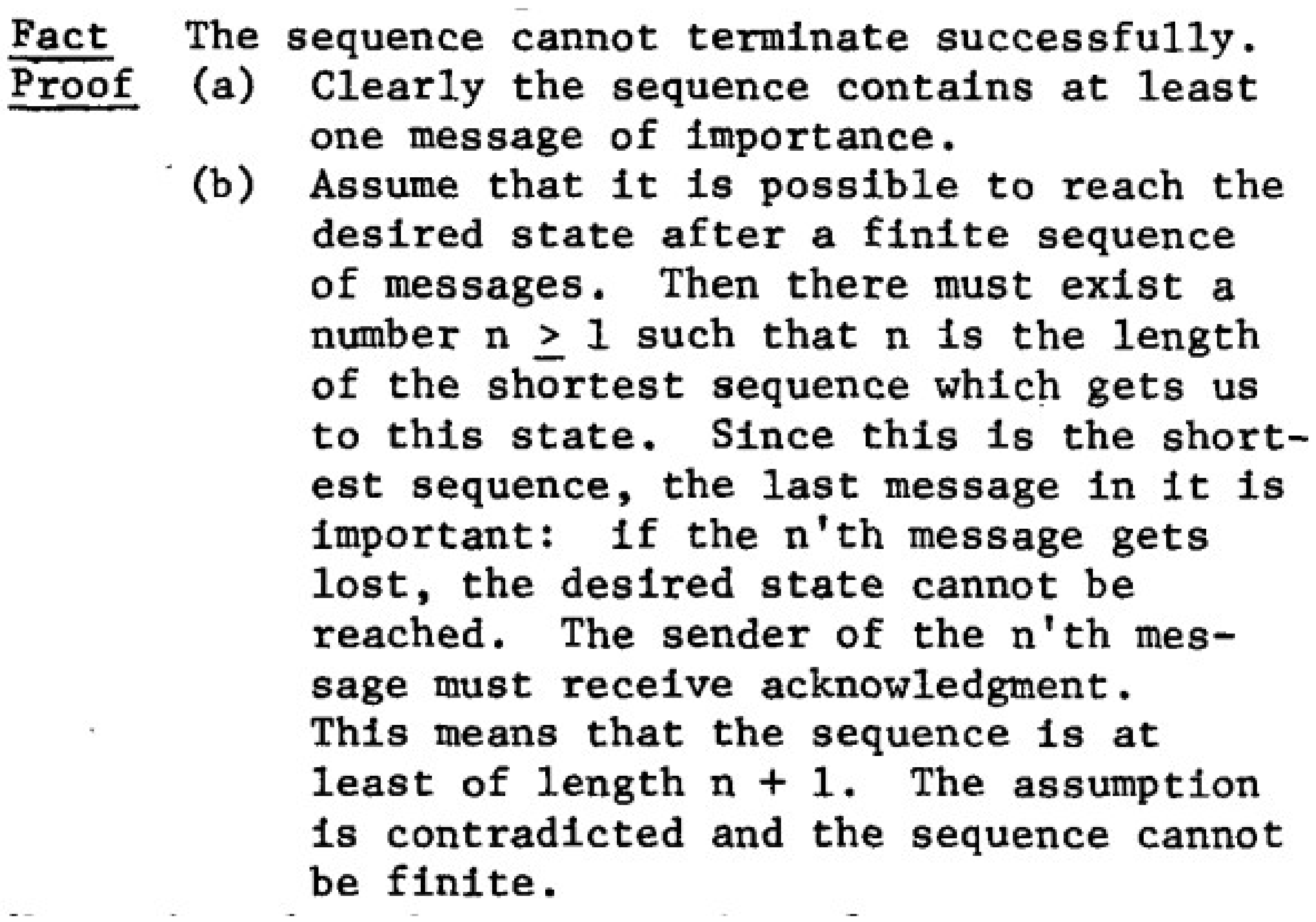
....



Note that the needs of both parties are quite reasonable. They simply want to reach a state where

(1) The original message (i.e., the plan of action) is successfully delivered, and

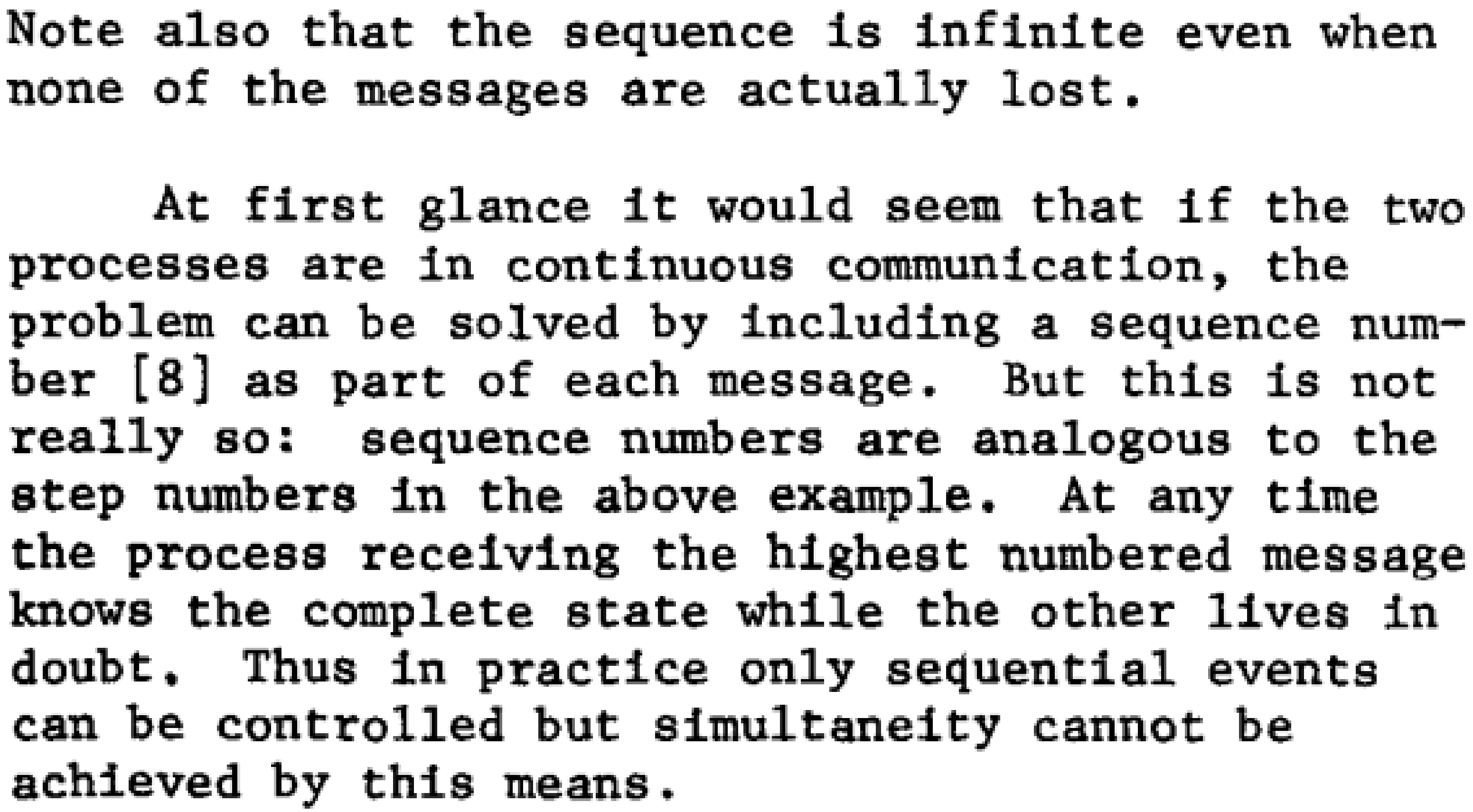
(2) Both parties know that they are in mutual agreement that (1) occurred.



Fact The sequence cannot terminate successfully.

Proof (a) Clearly the sequence contains at least one message of importance.

(b) Assume that it is possible to reach the desired state after a finite sequence of messages. Then there must exist a number $n \geq 1$ such that n is the length of the shortest sequence which gets us to this state. Since this is the shortest sequence, the last message in it is important: if the n'th message gets lost, the desired state cannot be reached. The sender of the n'th message must receive acknowledgment. This means that the sequence is at least of length $n + 1$. The assumption is contradicted and the sequence cannot be finite.

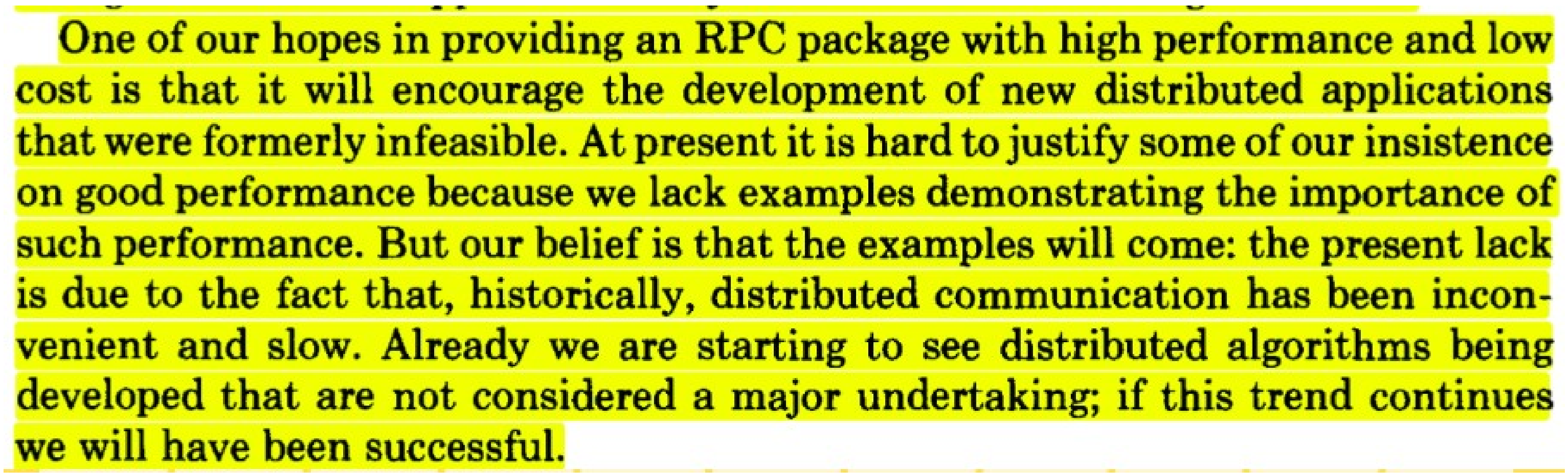


Note also that the sequence is infinite even when none of the messages are actually lost.

At first glance it would seem that if the two processes are in continuous communication, the problem can be solved by including a sequence number [8] as part of each message. But this is not really so: sequence numbers are analogous to the step numbers in the above example. At any time the process receiving the highest numbered message knows the complete state while the other lives in doubt. Thus in practice only sequential events can be controlled but simultaneity cannot be achieved by this means.

RPC

- RPC is not rocket science
  - But when you dig into the exact semantics, it is complicated
- However, it’s how much of distributed computing is done...
  - Java, Go, Python, Rust, .NET…
  - NFS, SunRPC, D-Bus, SOAP, WCF, DCOM, Google’s protobufs, Google Web toolkit



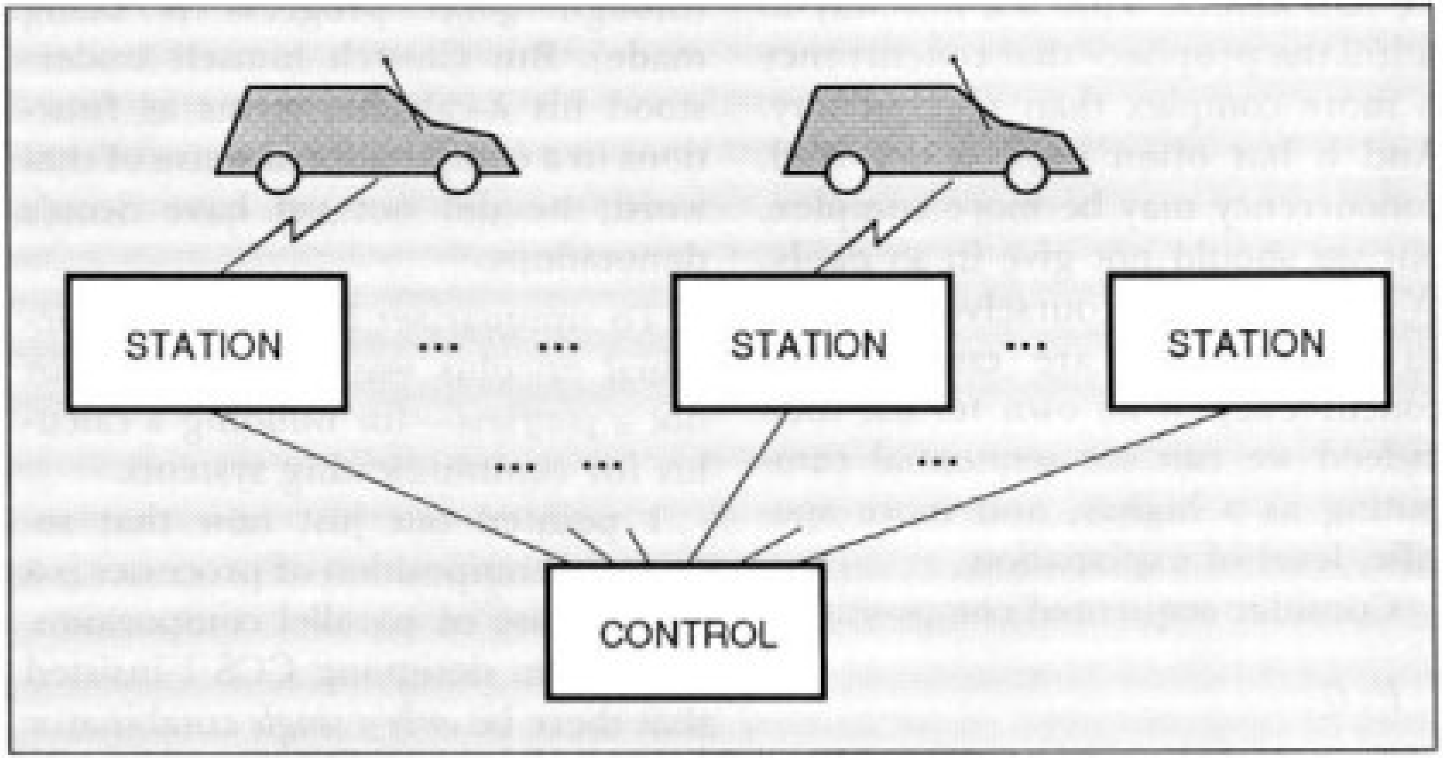
One of our hopes in providing an RPC package with high performance and low cost is that it will encourage the development of new distributed applications that were formerly infeasible. At present it is hard to justify some of our insistence on good performance because we lack examples demonstrating the importance of such performance. But our belief is that the examples will come: the present lack is due to the fact that, historically, distributed communication has been inconvenient and slow. Already we are starting to see distributed algorithms being developed that are not considered a major undertaking; if this trend continues we will have been successful.

Why not just use RPC all the time?

Disadvantages to RPC

- Multicast and broadcast are not well supported
- Caller blocks until they get a response, unless they fork a thread but then they need to again think about concurrency issues
- What if a process is physically moving?

Can we use semaphores, mutexes, RPC, *etc.* for this?



https://dl.acm.org/doi/pdf/10.1145/151233.151240

Coming up...

- Message passing