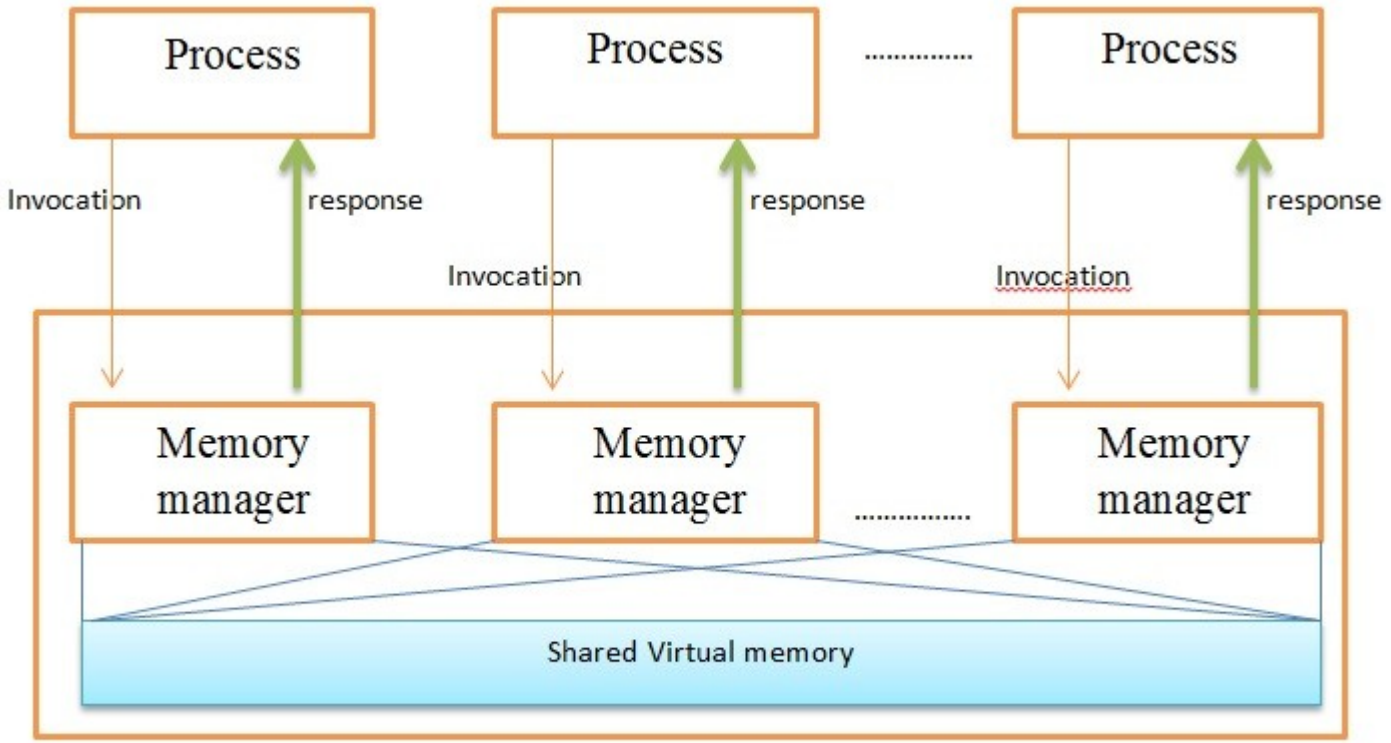
Distibuted Shared Memory and Remote Procedure Calls

CSE 536 Spring 2024
jedimaestro@asu.edu

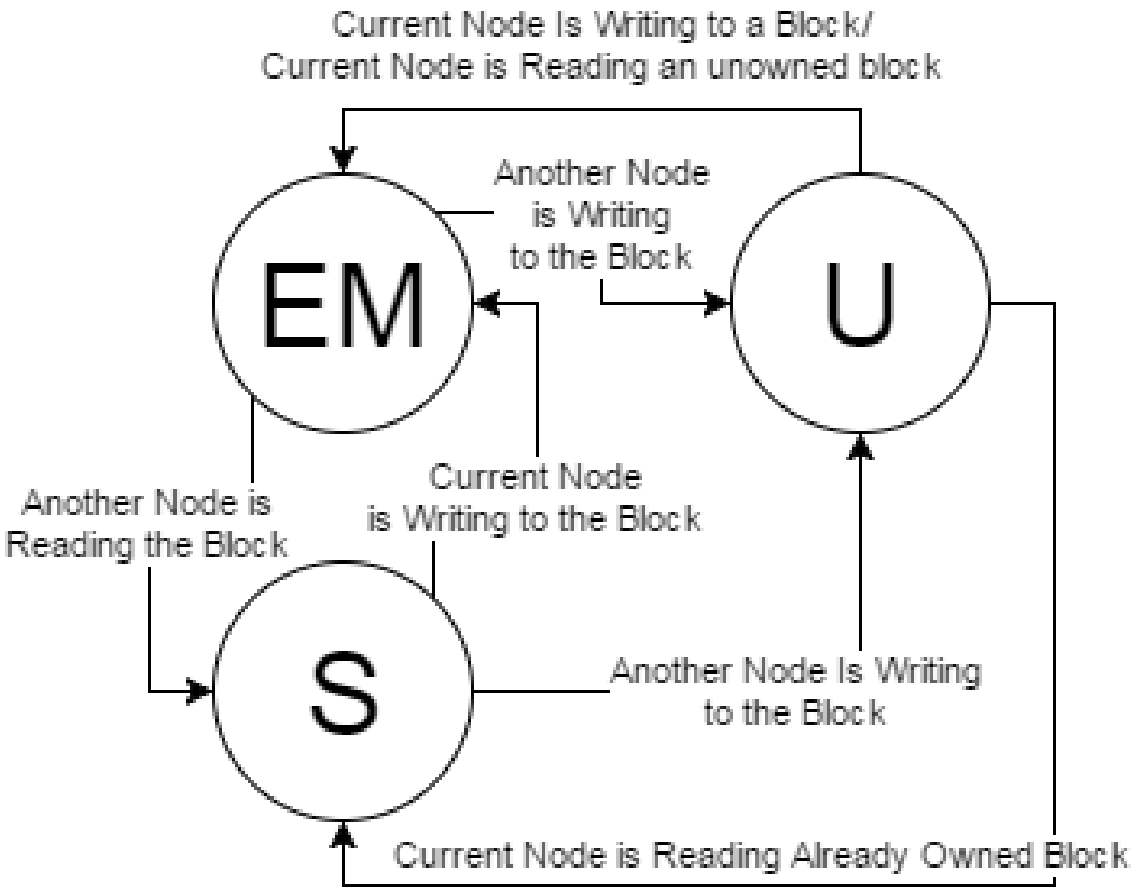
Outline

- Distributed Shared Memory
  - Consistency models
- Remote Procedure Calls

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



Distributed shared memory

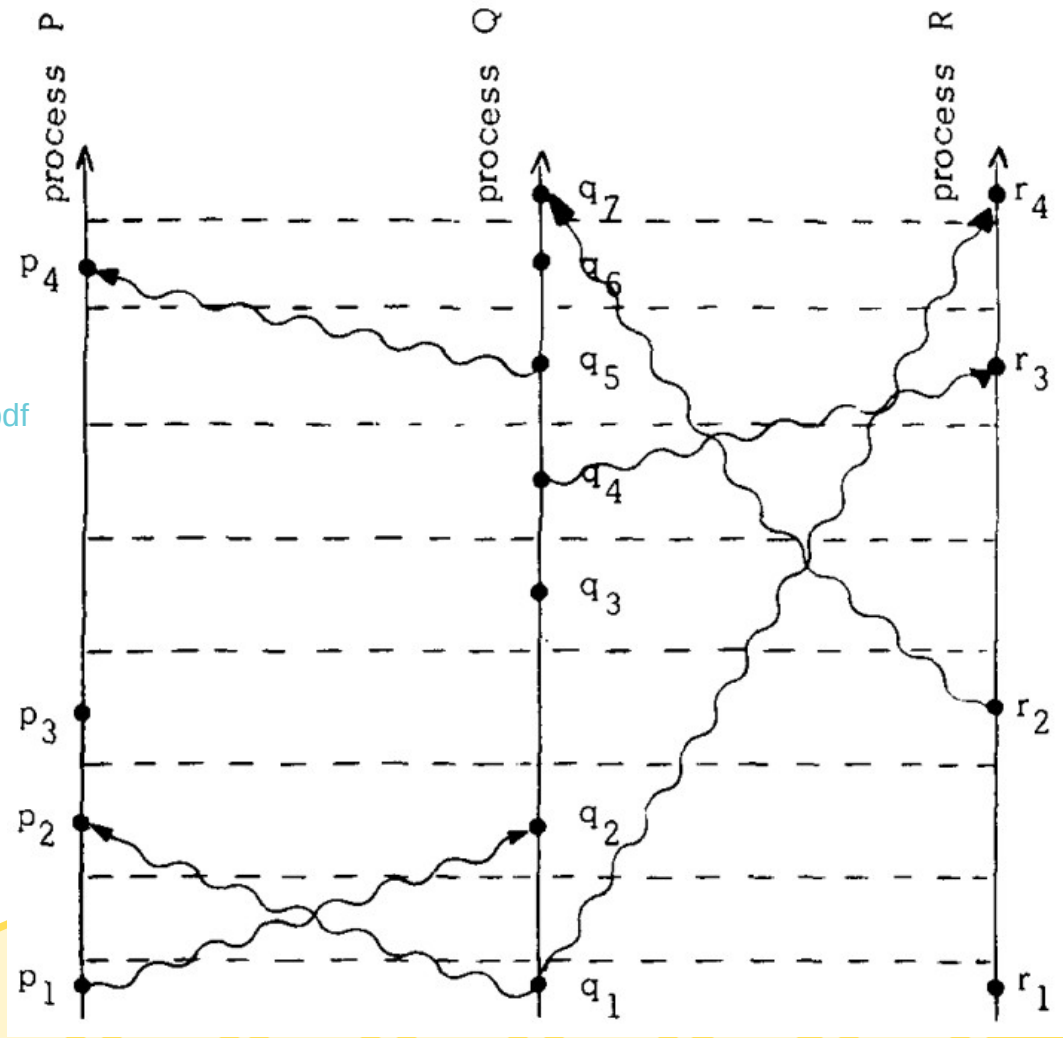

Current Node Is Writing to a Block/
Current Node is Reading an unowned block
Another Node
is Writing
to the Block
EM
U
Another Node is
Reading the Block
Current Node
is Writing to the Block
S
Another Node Is Writing
to the Block
Current Node is Reading Already Owned Block

Consistency models

- Strict consistency
  - All reads and writes in the same order for every process
  - More of a thought experiment
- 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, others...

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

Fig. 3.



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
  - Heavily used in practice
  - Caller or callee can crash, doesn’t break everything
- https://jedcrandall.github.io/courses/cse536spring2024/birrell842.pdf

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

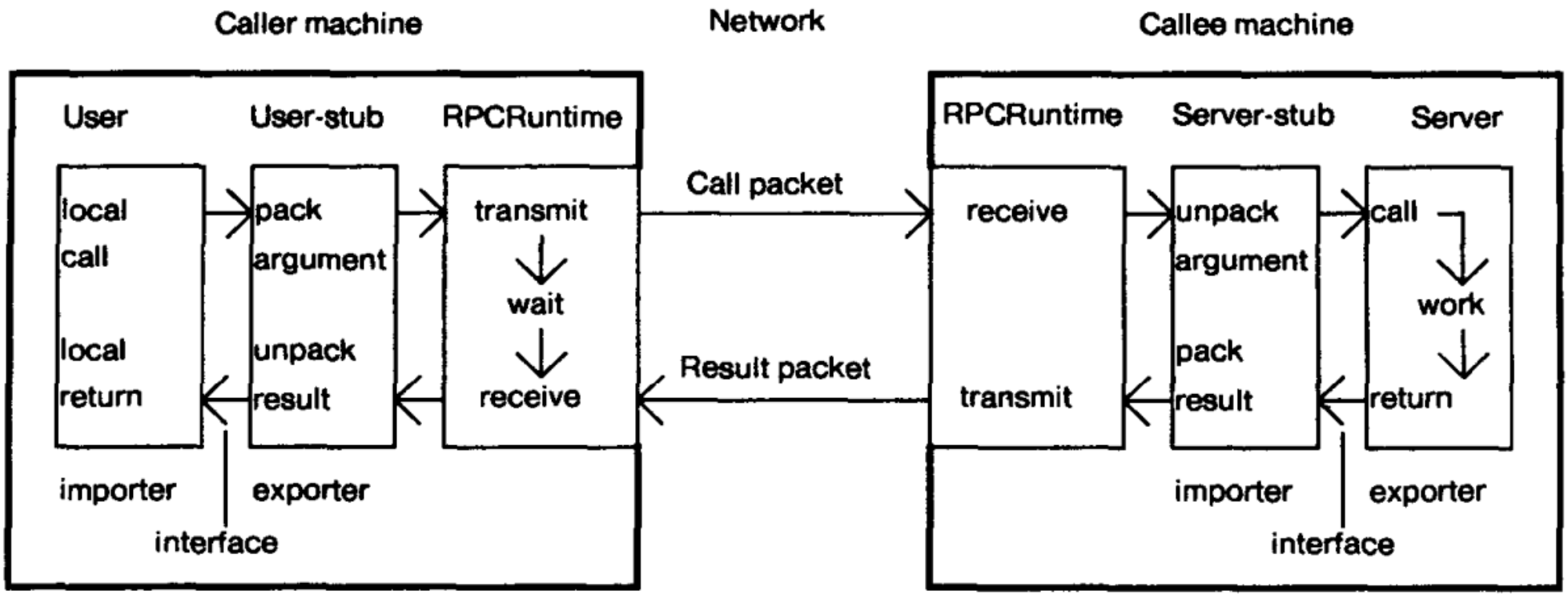


Fig. 1. The components of the system, and their interactions for a simple call.

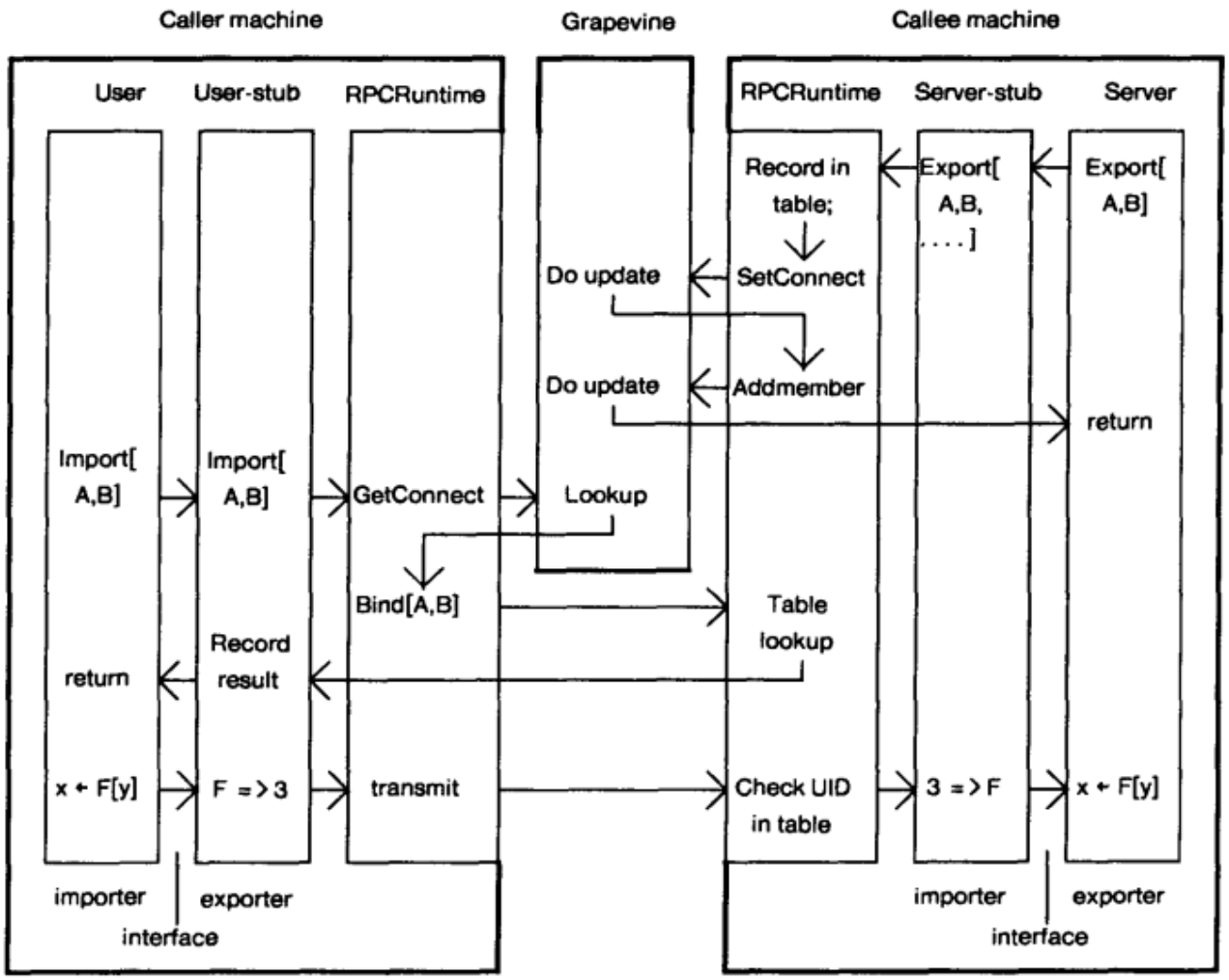


Fig. 2. The sequence of events in binding and a subsequent call. The callee machine exports the remote interface with type A and instance B. The caller machine then imports that interface. We then show the caller initiating a call to procedure F, which is the third procedure of that interface. The return is not shown.

2. BINDING

There are two aspects to binding which we consider in turn. First, how does a client of the binding mechanism specify what he wants to be bound to? Second,

ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984

how does a caller determine the machine address of the callee and specify to the callee the procedure to be invoked? The first is primarily a question of *naming* and the second a question of *location*.

2.1 Naming

The binding operation offered by our RPC package is to bind an importer of an interface to an exporter of an interface. After binding, calls made by the importer invoke procedures implemented by the (remote) exporter. There are two parts to the name of an interface: the *type* and the *instance*. The type is intended to specify, at some level of abstraction, which interface the caller expects the callee to implement. The instance is intended to specify which particular implementor of an abstract interface is desired. For example, the type of an interface might

Terminology

- Marshalling (not in the paper, but implied).. packing and unpacking (unmarshalling) the parameters
  - Necessary because of differences in machines, representations

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.

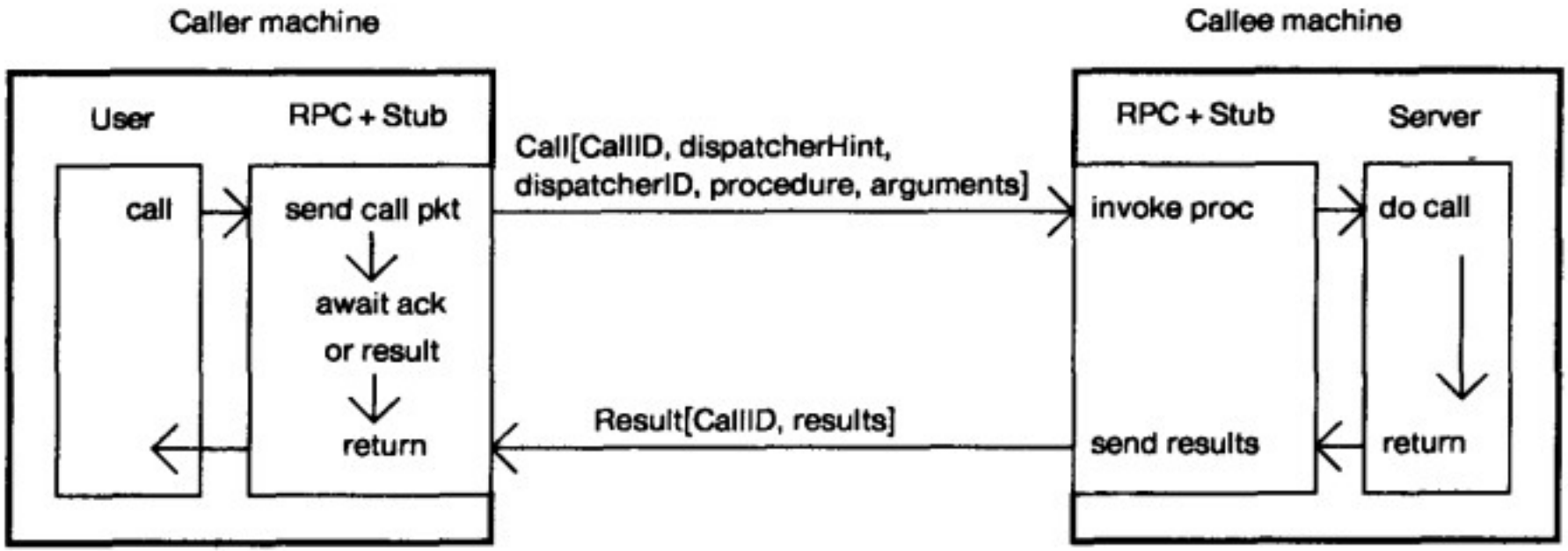


Fig. 3. The packets transmitted during a simple call.

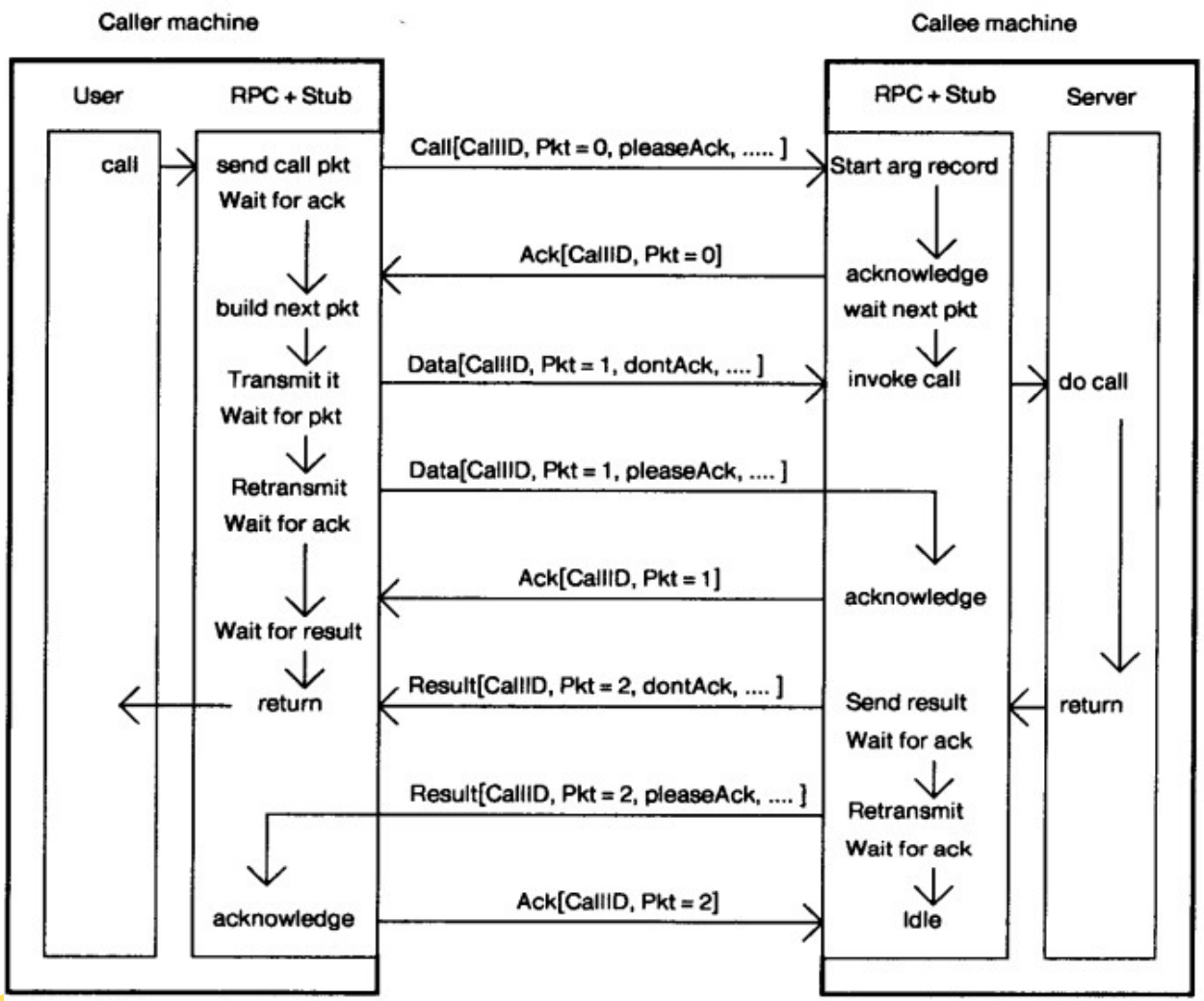


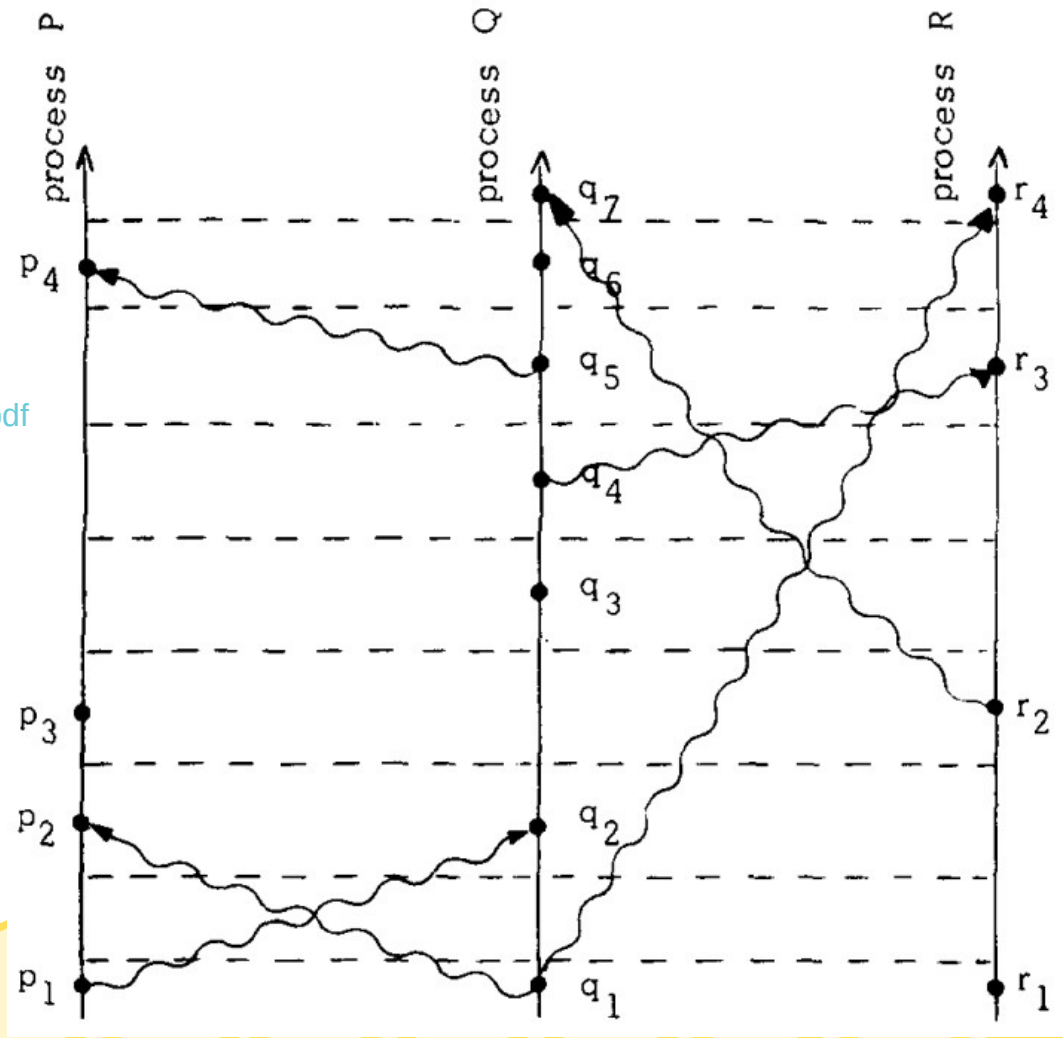
Fig. 4. A complicated call. The arguments occupy two packets. The call duration is long enough to require retransmission of the last argument packet requesting an acknowledgment, and the result packet is retransmitted requesting an acknowledgment because no subsequent call arrived.

Where does the “this solves the concurrency problem” part come?

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

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

Fig. 3.



We are still in the early stages of acquiring experience with the use of RPC and certainly more work needs to be done. We will have much more confidence in the strength of our design and the appropriateness of RPC when it has been used in earnest by the projects that are now committing to it. There are certain circumstances in which RPC seems to be the wrong communication paradigm. These correspond to situations where solutions based on multicasting or broadcasting seem more appropriate [2]. It may be that in a distributed environment there are times when procedure calls (together with our language's parallel processing and coroutine facilities) are not a sufficiently powerful tool, even though there do not appear to be any such situations in a single machine.

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.

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

RPC

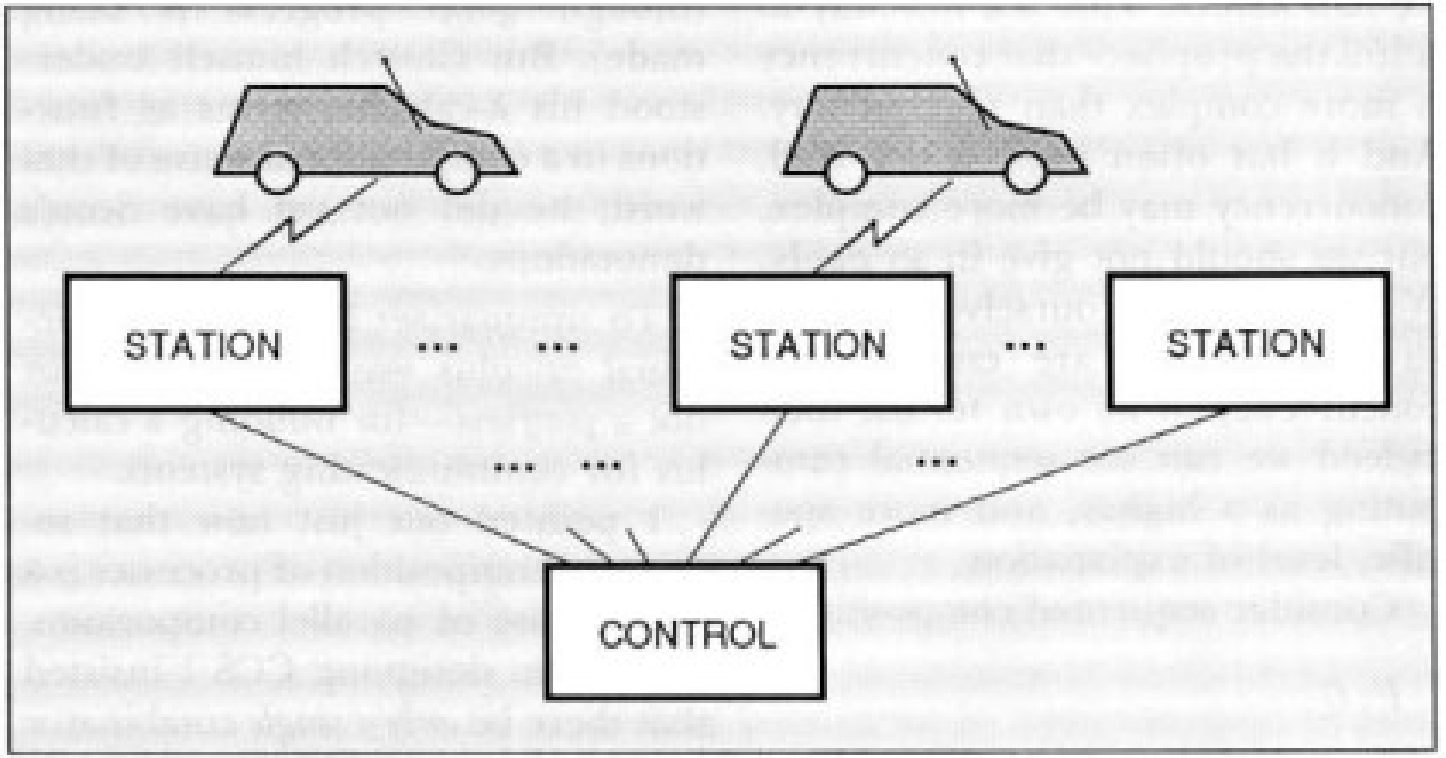
- RPC is not rocket science
- However, it’s how most of the world does distributed computing
  - Java, Go, Python, Rust, .NET…
  - NFS, SunRPC, D-Bus, SOAP, WCF, DCOM, Google’s protobufs, Google Web toolkit
- Do you think HTTP GET and POST requests are RPC?

Why not just use RPC all the time?

Disadvantages to RPC

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

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



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

Coming up...

- Message passing