Duke Systems

CPS 310
Servers and concurrency

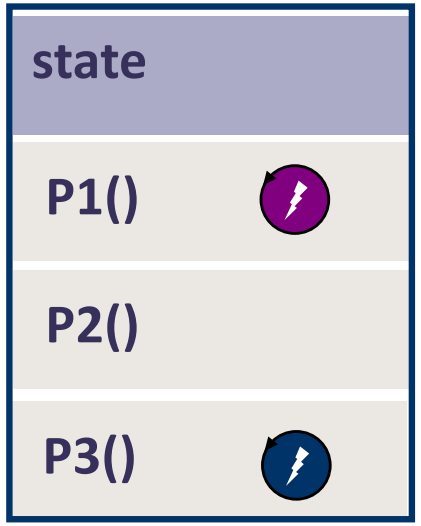
Jeff Chase

Duke University

From objects to servers

Modular atomic objects

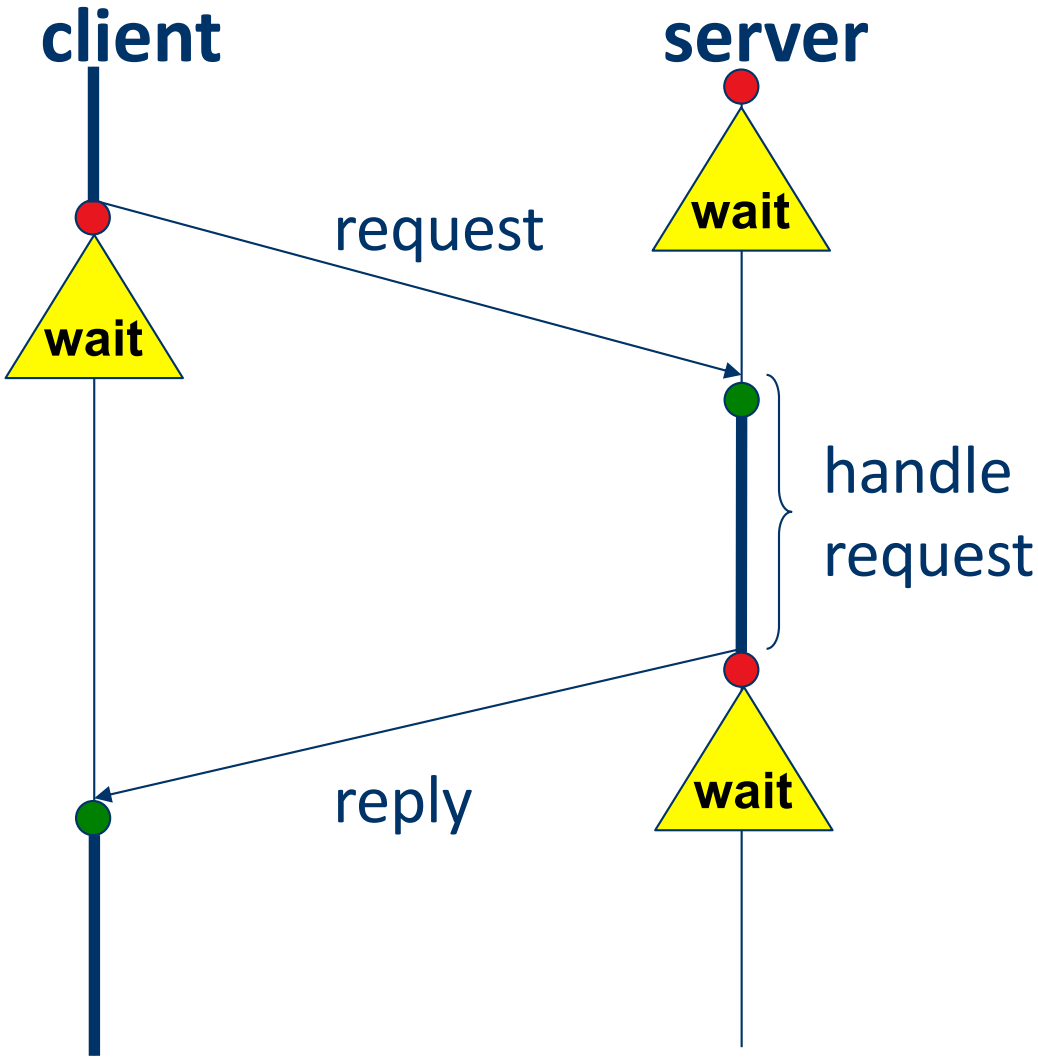
- A set of procedures/methods and API that defines how threads call/invoke them.
- Encapsulated/isolated: state accessed (only) by methods.
- Threads invoke API→ concurrency inside.
- "Atomic"→ internal concurrency control.



Servers are objects whose API calls are implemented with message exchanges.

- Calling threads are independent **clients**.
- Run anywhere; interact over a network.
- Protection boundary: machine+process

Request/reply messaging



Client initiates.
Server accepts.
Client waits.
Server replies.

Remote Procedure Call (RPC) is one common example of this pattern.

The Web is another.

Today many services run “RPC over HTTP”, e.g., REST, gRPC.

gRPC

grpc.io

gRPC can help make connecting, operating and debugging distributed systems as easy as making local function calls; the framework handles all the complexities normally associated with enforcing strict service contracts, data serialization, efficient network communication, authentications and access control, distributed tracing and so on. gRPC along with **protocol buffers** enables loose coupling, engineering velocity, higher reliability and ease of operations. Also, gRPC allows developers to write service definitions in a language-agnostic spec and generate clients and servers in multiple languages.

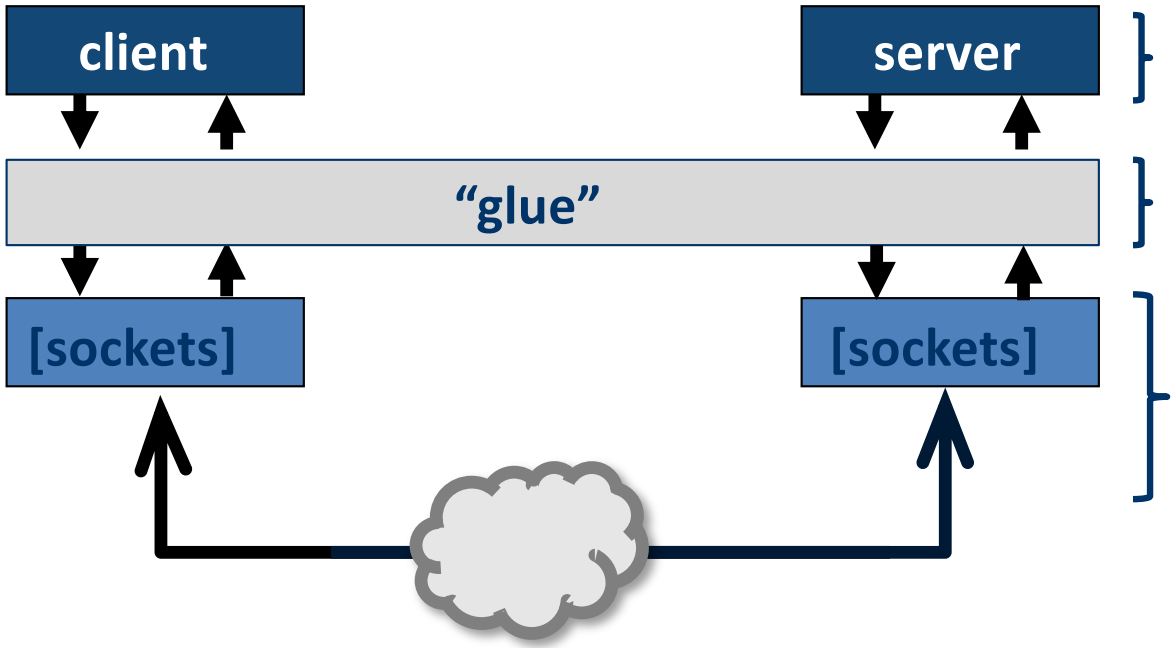
2020

Google Cloud

https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments?m=1

Remote Procedure Call (RPC)

- "RPC is a canonical structuring paradigm for client/server request/response services."
- First saw wide use in 1980s client/server systems for workstation networks (e.g., Network File System).
- Build it over TCP or over raw messaging, or…



Humans focus on getting this code right.

Auto-generate **stub** code from API spec (IDL).

This code is "canned", independent of the specific application.

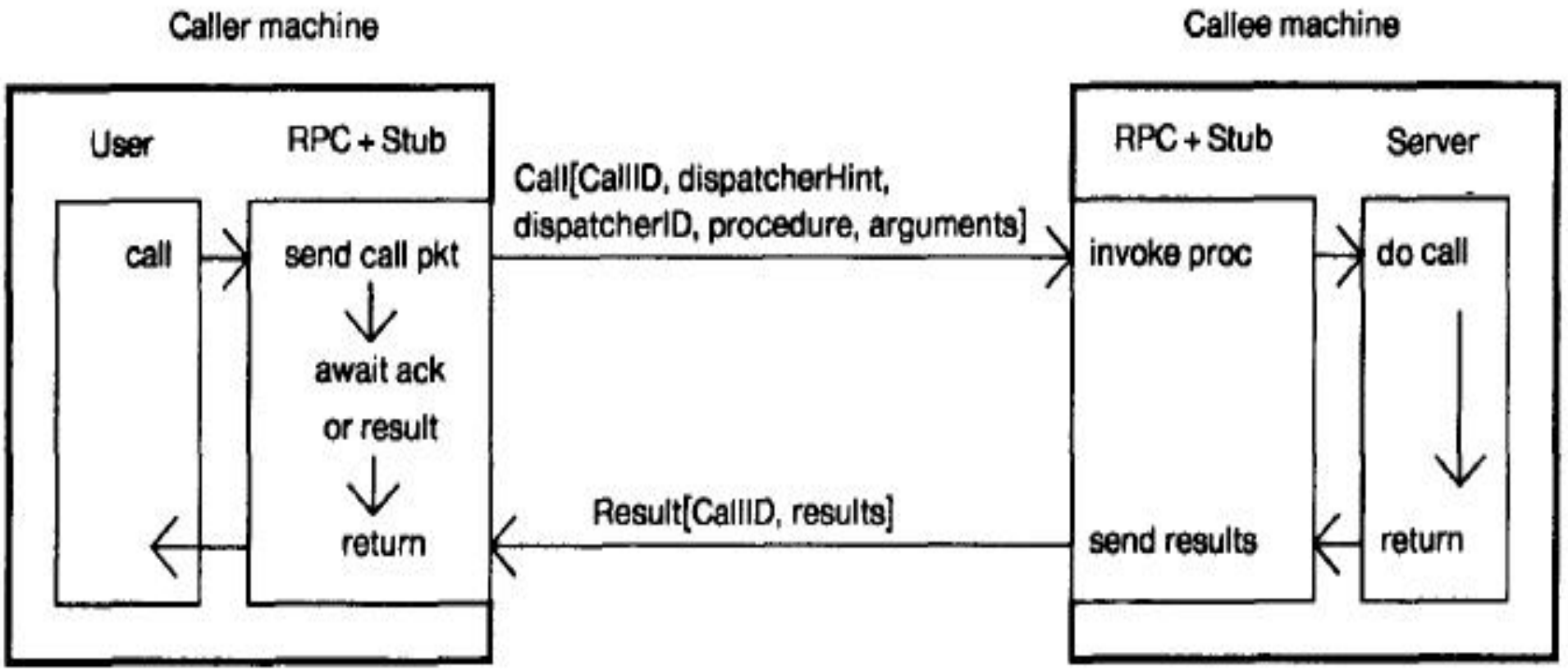
Implementing Remote Procedure Calls

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

Remote procedure calls (RPC) appear to be a useful paradigm for providing communication across a network between programs written in a high-level language. This paper describes a package providing a remote procedure call facility, the options that face the designer of such a package, and the decisions we made. We describe the overall structure of our RPC mechanism, our facilities for binding RPC clients, the transport level communication protocol, and some performance measurements. We include descriptions of some optimizations used to achieve high performance and to minimize the load on server machines that have many clients.

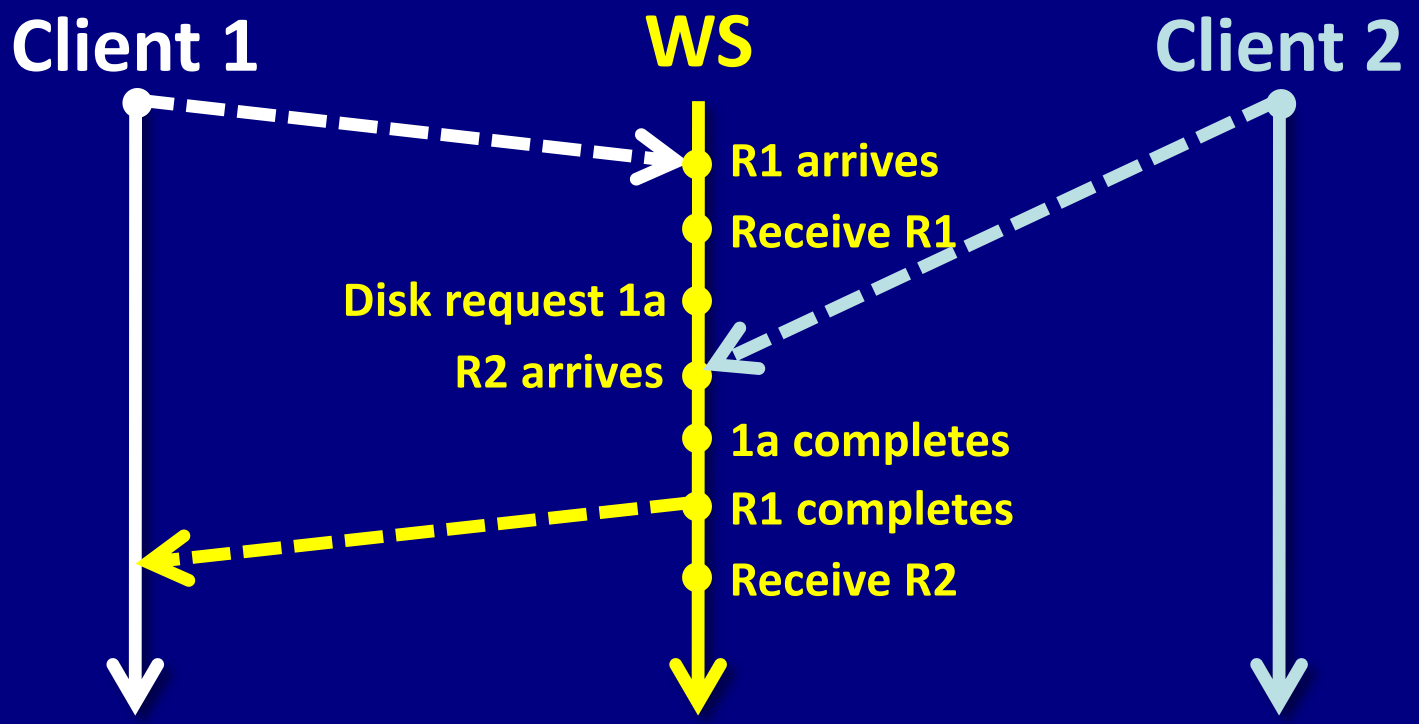
1984. ACM SIGOPS Hall of Fame paper
2700 citations

Simple RPC Diagram



Server (serial process)

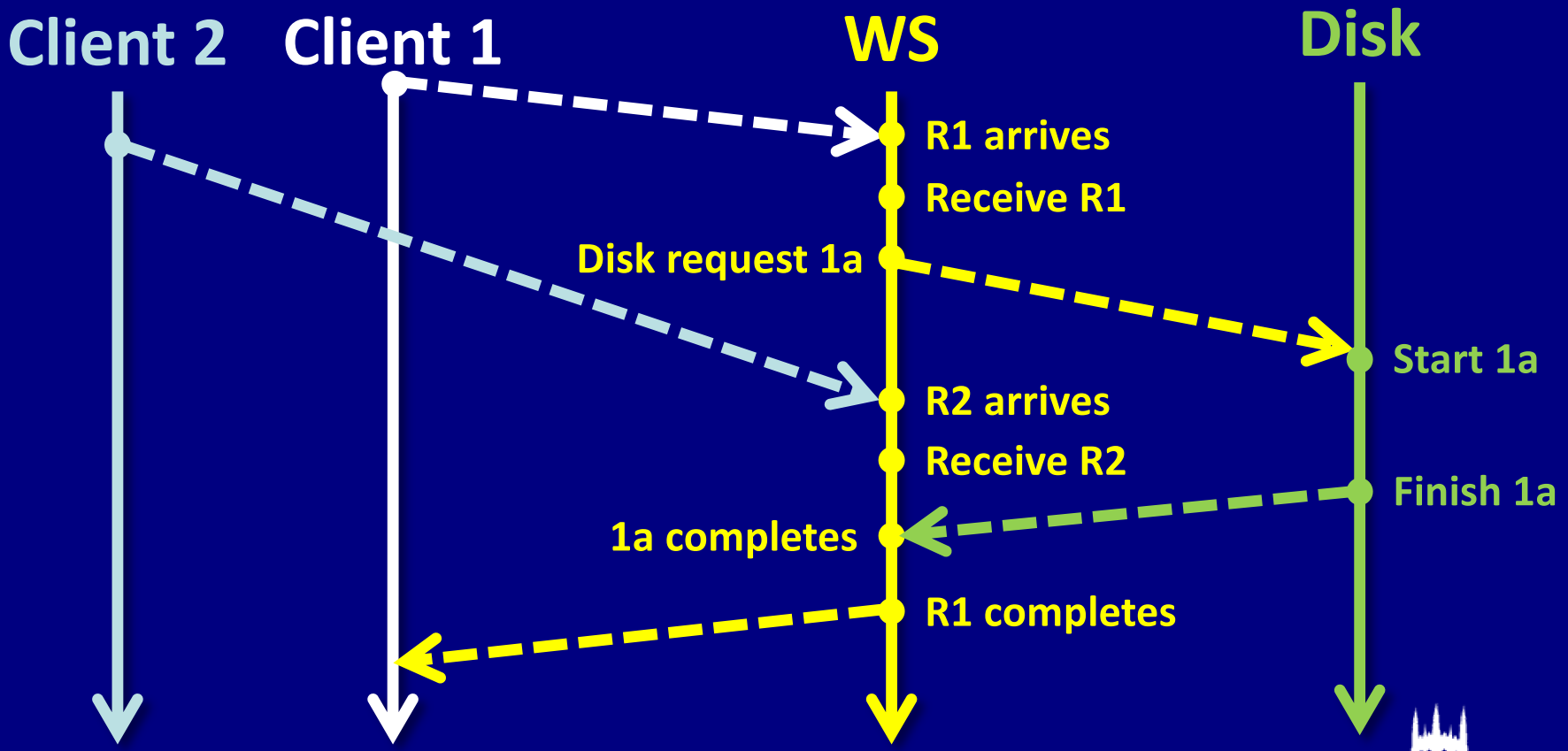
- Option 1: could handle requests serially



- Easy to program, but painfully slow (why?)

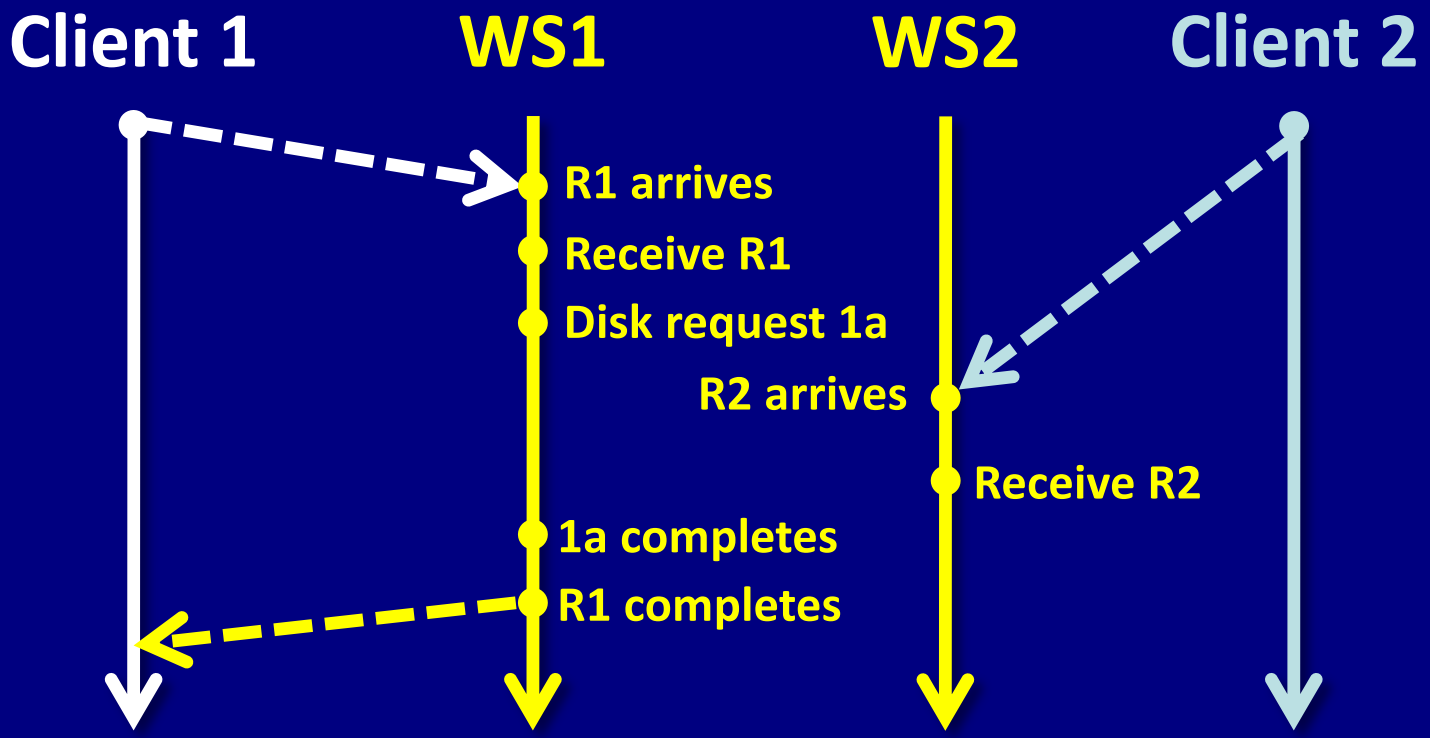
Server (event-driven)

- Option 2: use asynchronous I/O
- Fast, but hard to program (why?)



Server (multiprogrammed)

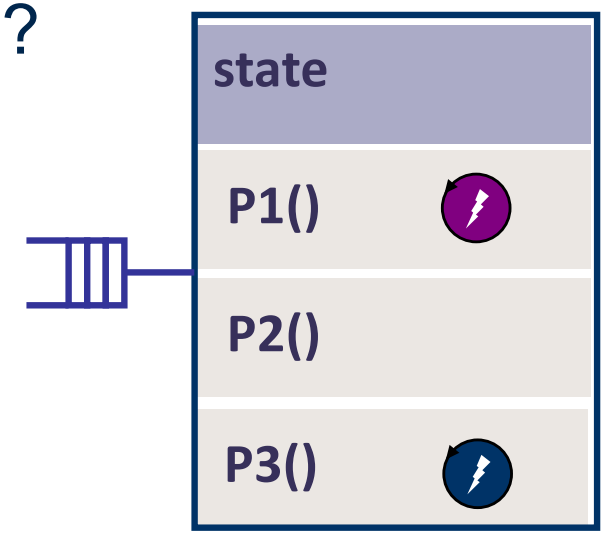
- Option 3: assign one thread per request



- Where is each request's state stored?

Multi-threaded server

- Multi-threaded server is a common design pattern.
  - Standard multi-threaded process
  - Bounded incoming request queue
- Why not processes instead of threads?
  - OK for “classic” Web servers
  - And other stateless servers
  - Isolated/contained
- Multi-threading is accepted now:
  - More comfortable for shared state
  - “Lightweight” concurrency, easy blocking



Threads and RPC

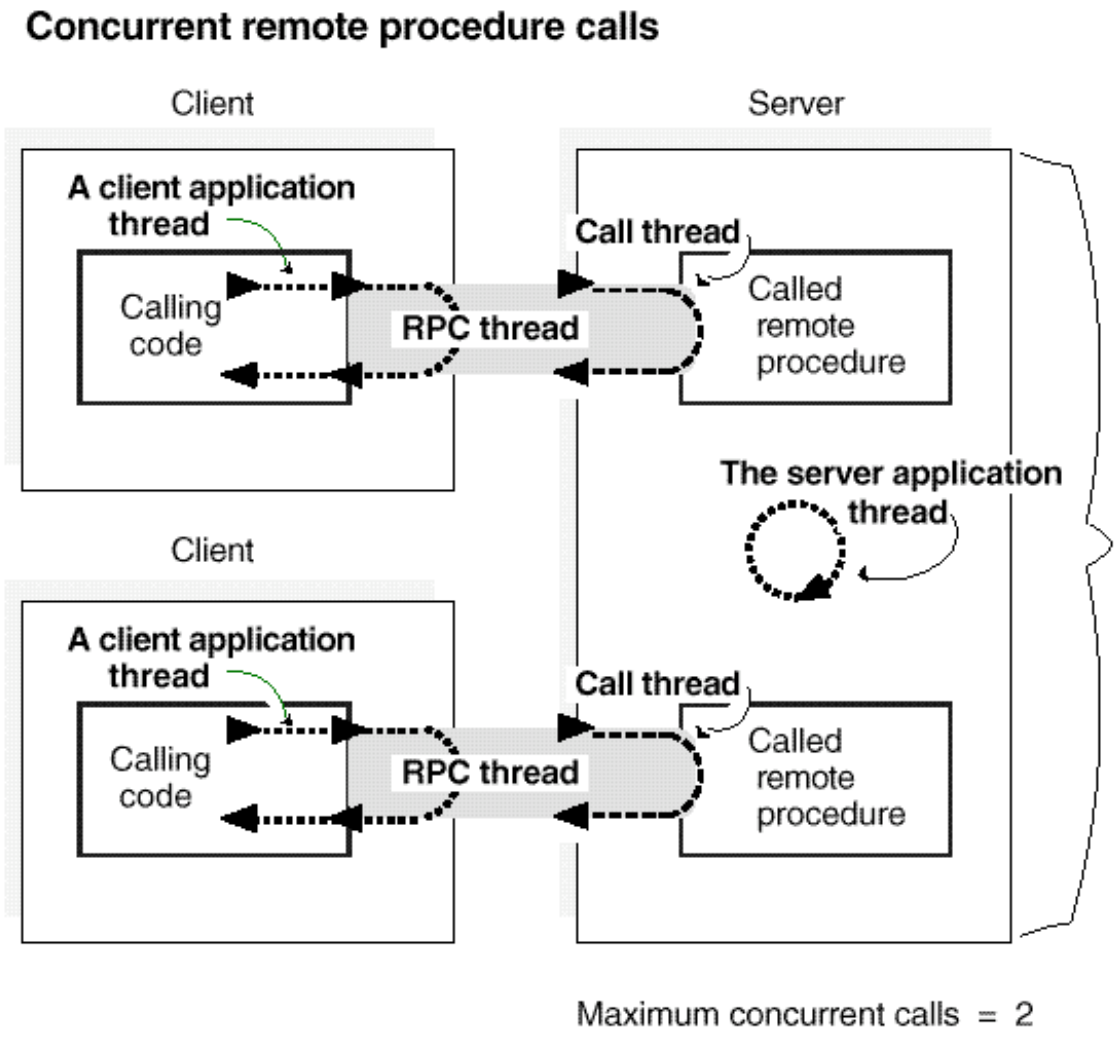


Figure 6-2 Concurrent Call Threads Executing in Shared Execution Context

[OpenGroup, late 1980s]

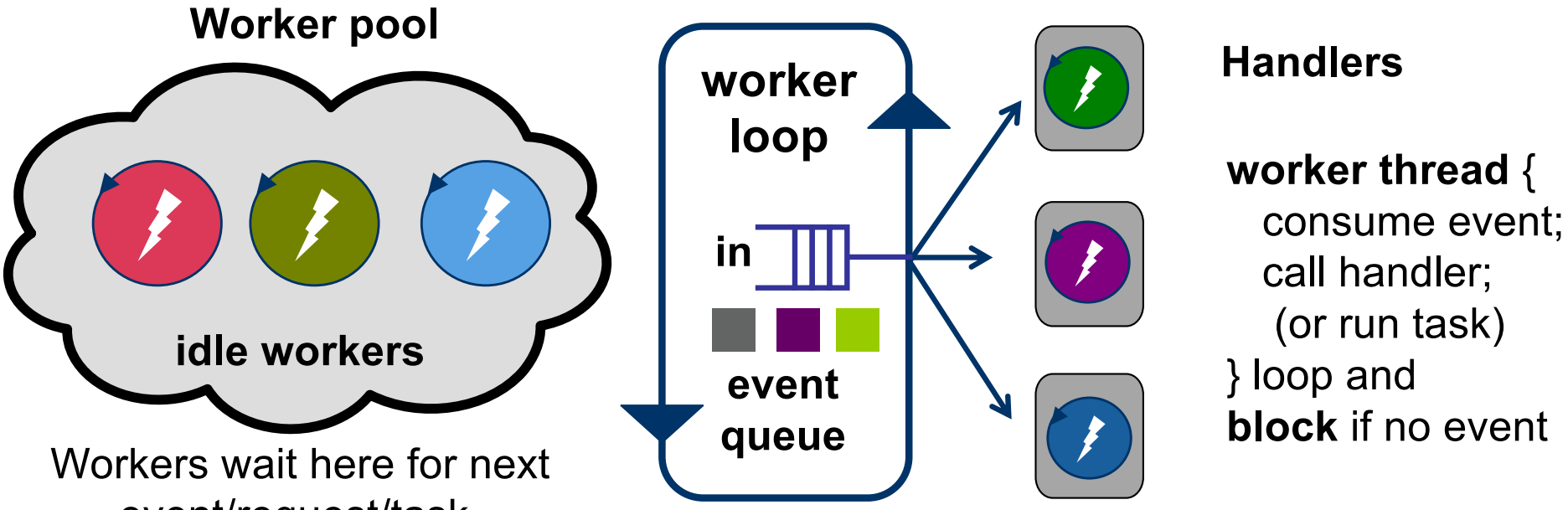
Q: How do we manage server "call threads"?

A: Create them as needed, and keep idle threads in a **thread pool**.

When an RPC call arrives, wake up an idle thread from the pool to handle it.

On the client, the client thread blocks until the server thread returns a response.

Thread pool (executor)

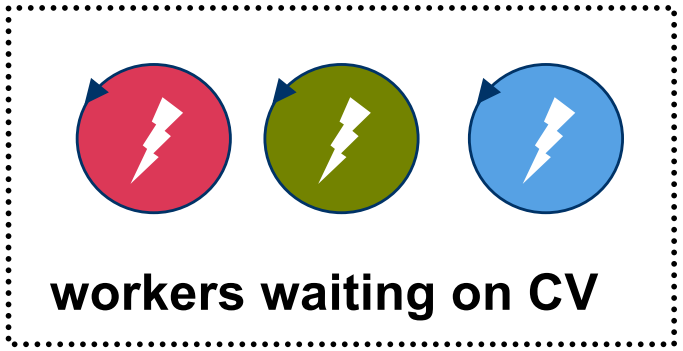


- Thread pool: a pattern for parallel programs and network servers.
- N workers can run in parallel on N cores: also called **WorkCrew**.
- Queue of incoming tasks—equivalent to events+handler calls.
- Any worker thread can call any handler or run any task.

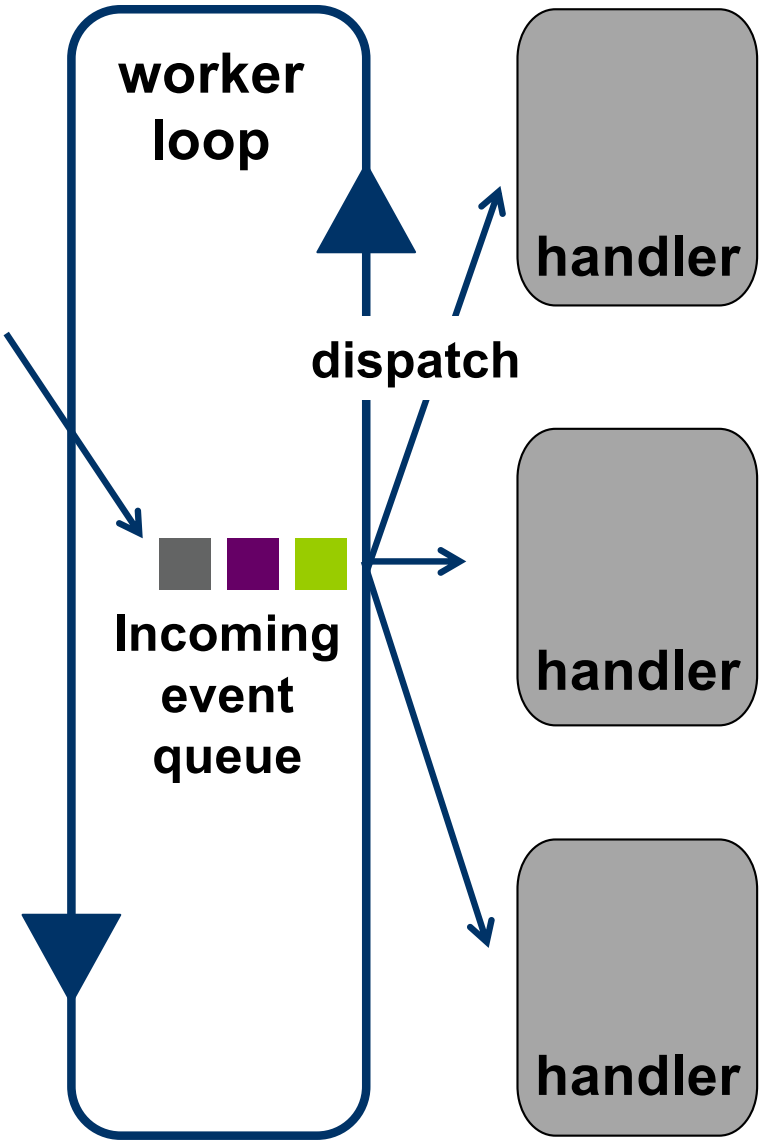
Event/request queue

Synchronize queue with a monitor: a mutex/CV pair.

Protect the event queue data structure itself with the mutex.



Workers **wait** on the CV if the event queue is empty. **Signal** the CV when a new event arrives. **Producer/consumer bounded buffer**.



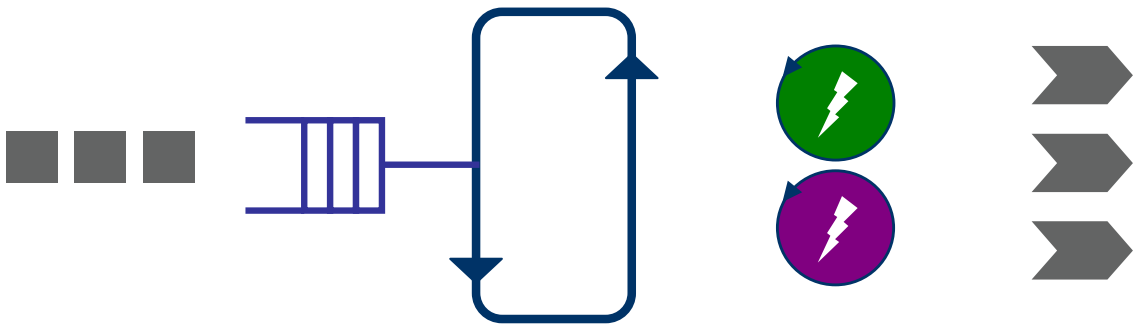
Handle one event, blocking as necessary.

When handler is complete, return to worker pool.

Ideal event poll API for thread pooling

Abstract poll(): a long time to get this right in real systems.

1. Delivers: returns exactly one event (message or notification), in its entirety, ready for service (dispatch).
2. Idles: Blocks iff there is no event ready for dispatch.
3. Consumes: returns each posted event at most once.
4. Combines: any of many kinds of events (a poll set) may be returned through a single blocking call to poll.
5. Synchronizes: may be shared by multiple processes or threads (→ handlers must be thread-safe as well).

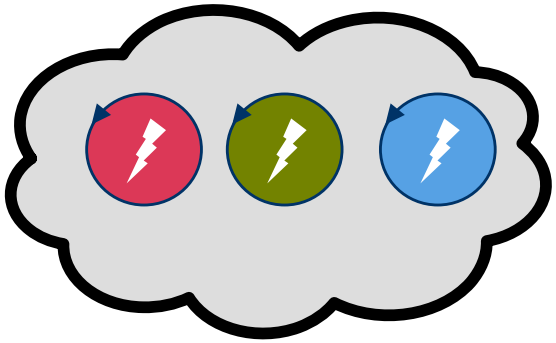


Managing load and concurrency

- **How many worker threads?**
- What if requests/tasks block?

  >N workers to keep N cores busy
- **What if request queue is full?**
  - More threads?
  - Flow control in the network?
  - Drop and say “try again later”?
  - Get a bigger server?
  - Shed load to another server?

  …



Magic elastic worker pool