Messages, RPC, Clients, and Servers

Daniel Andresen

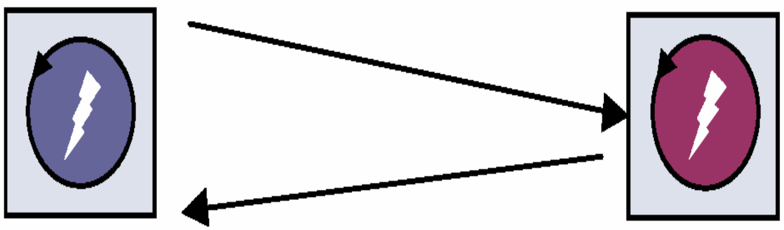
CIS520 – Operating Systems

Preview

1. Begin with IPC and messages, and use them as a starting point for a general discussion of system structuring alternatives.
    - request/response client/server interactions, server-structured systems, and "microkernels"
2. Introduce *Remote Procedure Call* (protected procedure call) as simple "syntactic sugar" for client/server communication.
    - messaging boundaries as module protection boundaries
3. Explore fundamental issues raised for systems and/or applications that are "communication-based" using messages or RPC.
    - managing communication endpoints: the *port* abstraction
    - the role of threads, vs. event-based structuring

IPC with message Send/Receive

A common and useful IPC abstraction: Generalized message *send* and *receive* primitives.

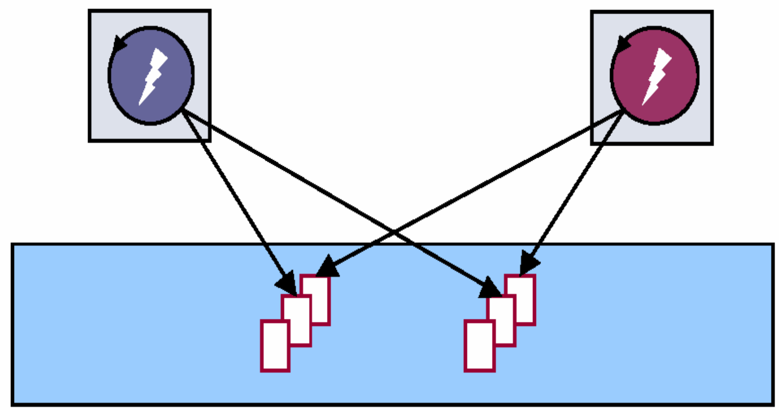


A messaging interface allows a process to send messages to a particular destination, **e.g.,**:

thread->send(data);

currentThread->receive(data);

Like pipes, messaging combines synchronization and data transfer.



Messages for a given destination are stored in a queue pending delivery. *Send* and *receive* are typically system calls, with message queues maintained by the kernel.

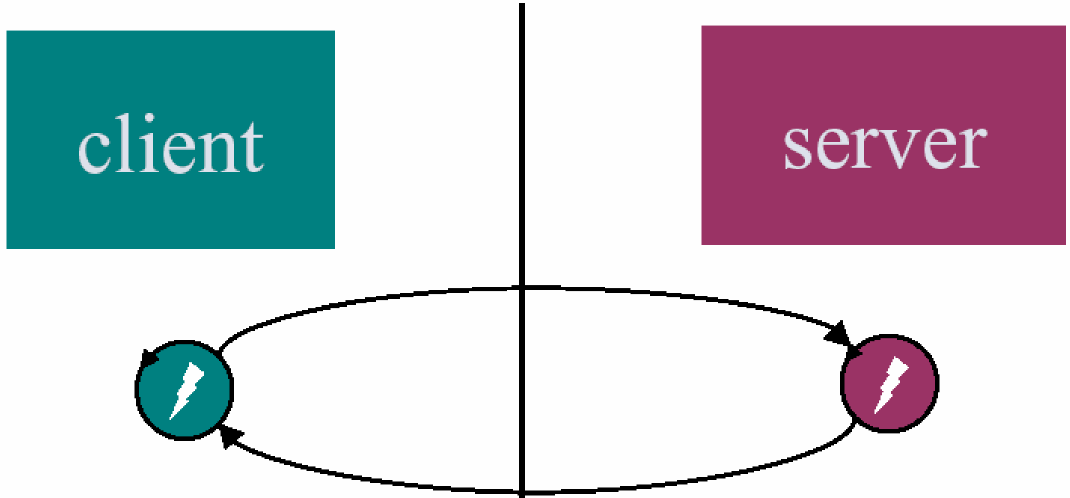
Like pipes, messaging combines synchronization and data transfer.

Issues for Message Send/Receive

- How to name the message destination?
  - e.g., send to a thread, connection, or “port”
- Message format/constraints
- Does the *receive* primitive block until a message arrives?
  - Or does the receiver poll for a message, or receive an interrupt.
- Does *send* block until the message is sent?
- How/where is the message queued awaiting the receiver?
- What if the receiver’s message queue is full?
- Is sending a message a privileged operation?
- How does a receiver identify the sender of a received msg?
- Can the receiver select the next msg from a specific sender?

Client/Server Request/Response

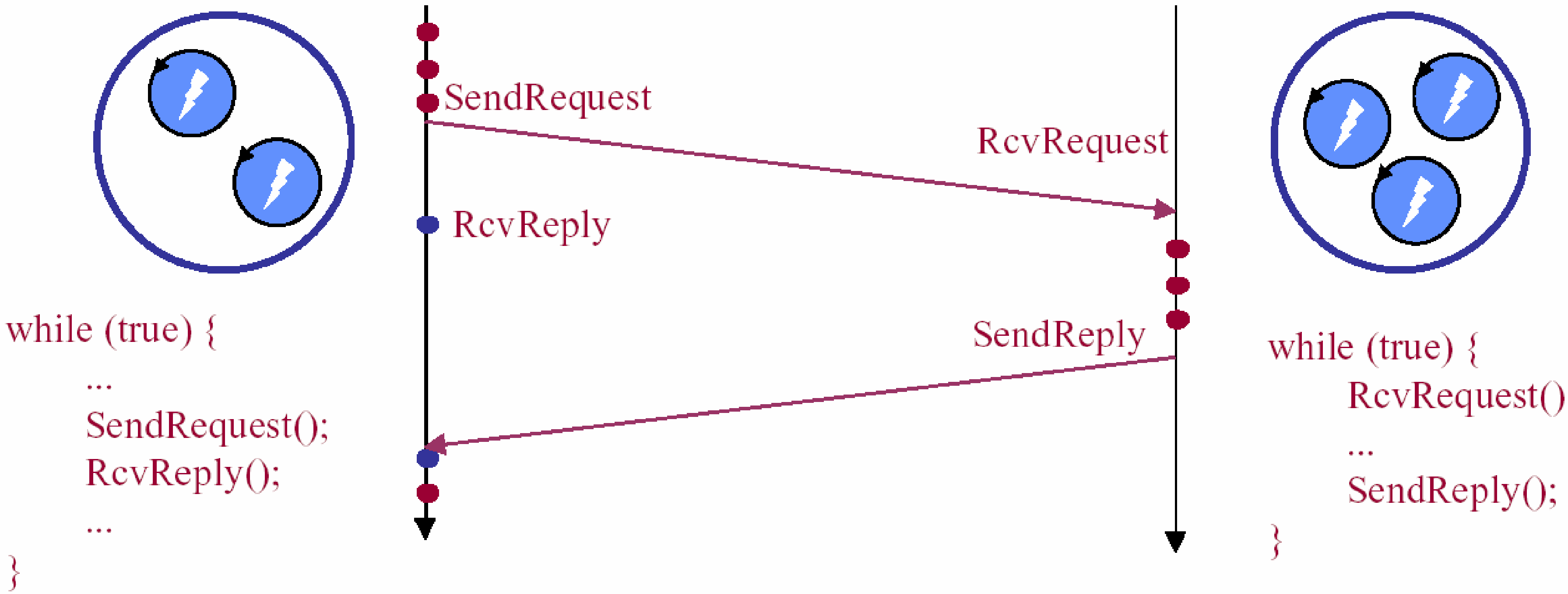
•One common style of messaging is for a server process to provide services to *client* processes on demand, using *request/response* message exchanges.




```
Thread* client;
....
client->send(request);
response = currentThread->receive();
...
```

```
while(systemActive) {
    currentThread->receive(request);
    handle the request
    requester->send(response);
}
```

Clients and Servers as Interacting Processes



Note the synergy with threads:

1. Client blocks until a reply is received.
   - Threads allow a client to issue concurrent requests.
2. Server waits for a request to arrive.
   - Threads allow a server to handle concurrent requests.

Messaging and Protection

- Like the kernel, the server is protected from its clients.
  - Address space isolation is preserved, so the client cannot corrupt the server's data.
  - The only way a client can cause code to run in the server is to send a message. The server decides how to validate and interpret each message.
- The client is also protected from the server, although it must rely on it to correctly perform the service.

  (Unlike the kernel, the server cannot access client memory.)
- *Protected servers may coordinate interactions among processes, manage system-critical data, or otherwise assume roles "typically" reserved for the operating system kernel.*

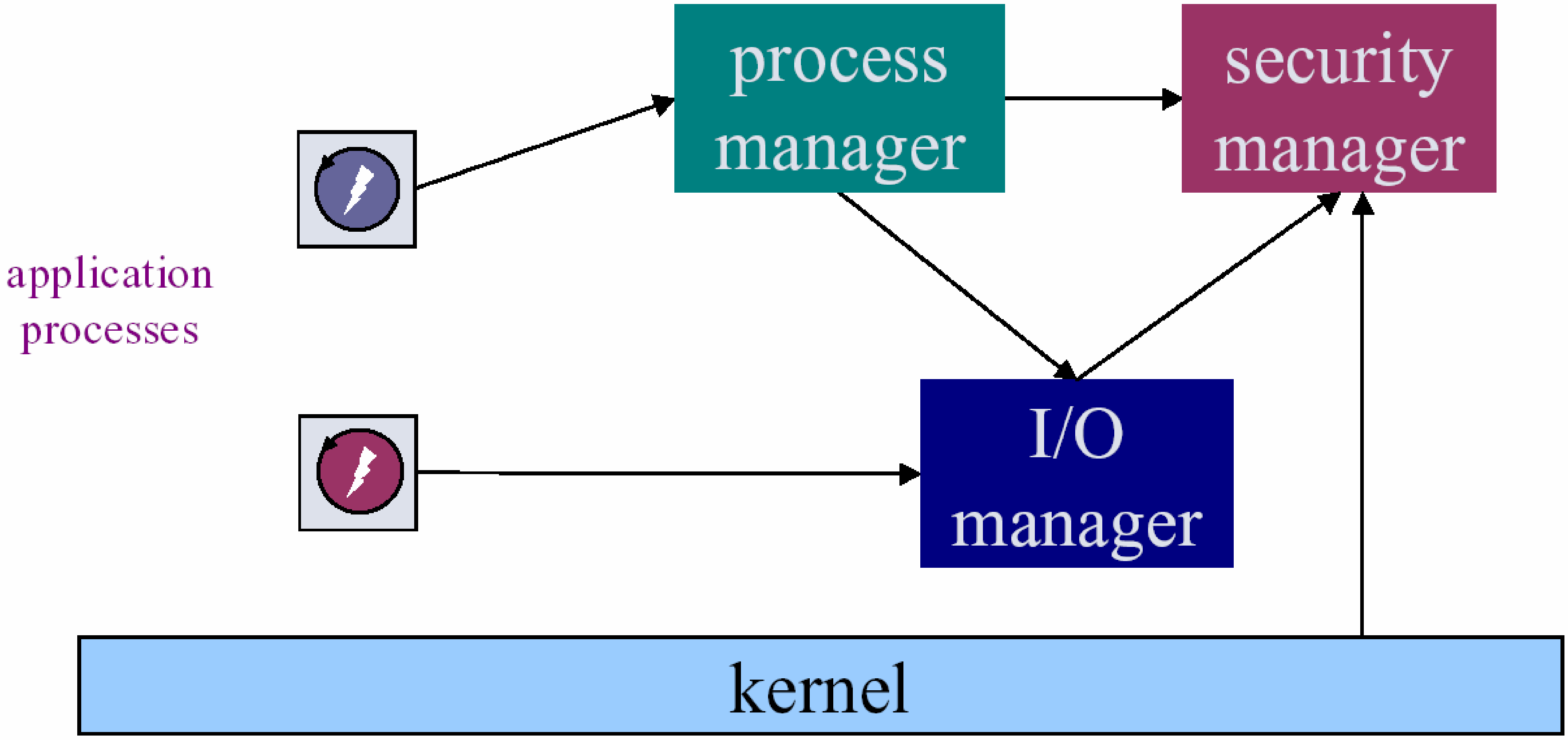
Reconsidering the Kernel Interface

- The kernel can be thought of as nothing more than a server; it is special only in that it runs in a protected hardware mode.
- Many of the services traditionally offered by the kernel can be supported outside of the kernel, in servers or in libraries.
- What features *must* be implemented in the kernel? Could we implement (say) the entire Unix interface as an application?
  - Why would we want to do such a thing?
- What are the advantages of supporting some OS feature in a server rather than directly in the kernel? What are the costs?
- How would we design a kernel interface that is powerful enough to implement multiple OS "personalities" as servers?
- *The kernel interface is not the programming interface!*

Server-Structured Systems and Microkernels

- A number of systems have been structured as collections of servers running above a minimal kernel (“*microkernel*”).
- **Microkernel** provides, e.g.,
  - basic threads and scheduling, IPC, virtual address spaces, and device I/O primitives.
- Kernel is hoped to be smaller, more reliable, and more secure.
- Policies (e.g., security) may be implemented outside of the kernel.
- Operating system “personalities” (e.g., Unix or Windows) may be implemented as servers.
  - OS may have multiple personalities and policies, with new OS features and APIs added on-the-fly.
- The performance of server-structured systems is determined largely by the efficiency of the messaging primitives.

Illustration of a Server-Structured Kernel



Some Microkernel History

The microkernel philosophy evolved in the mid-1980s as a reaction to the increasing complexity of Unix kernels.

- V system [Cheriton]: kernel is a “software backplane”
  - advent of LAN networks: V supports distributed systems, and mirrors their structure internally (decomposed)
- Mach: designed as a modern, portable, reconfigurable Unix
  - improve portability/reliability by “minimizing” kernel code
  - support multiple “personalities”; isolate kernel from API changes
  - support multiprocessors via threads and extensible VM system

Microkernels are widely viewed as having “failed” today, but the key ideas (and code) survive in modern systems (NT).

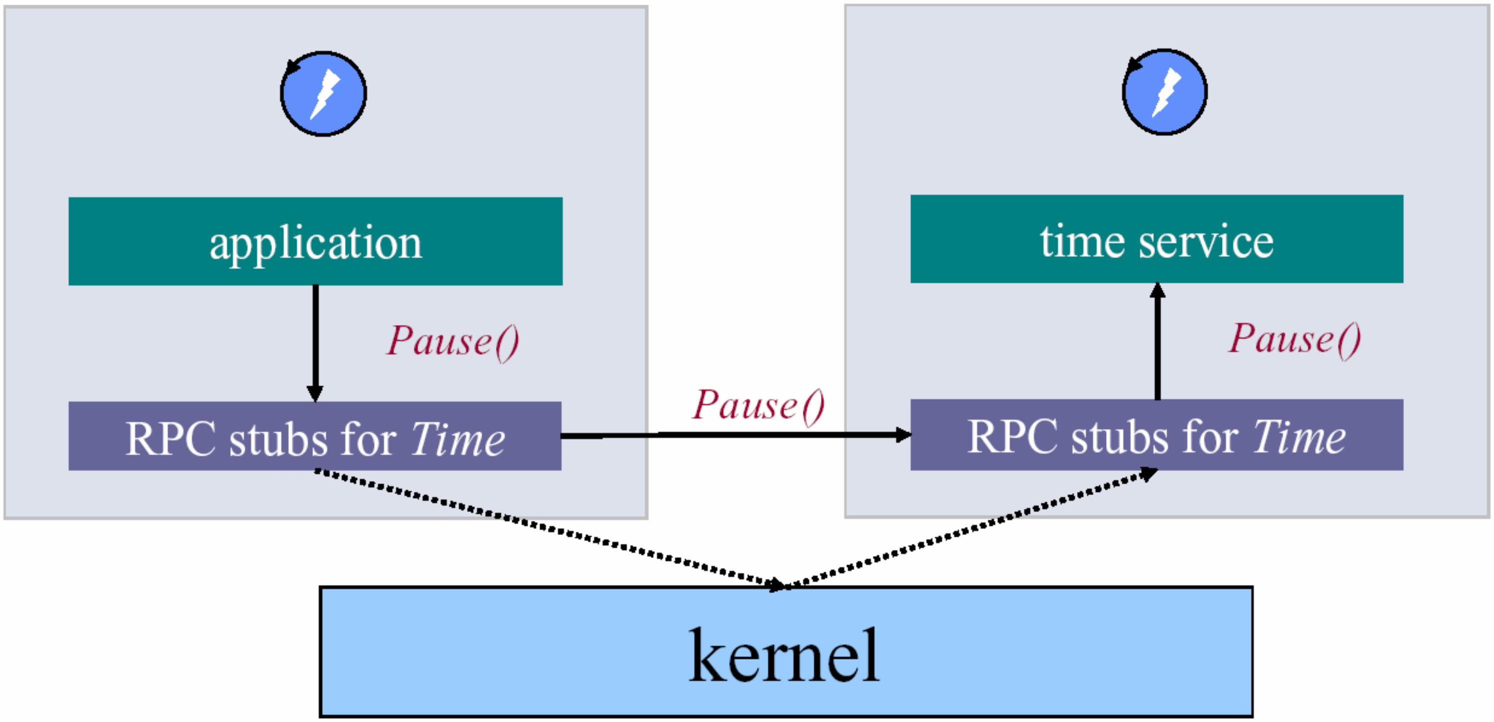
Beyond messaging - RPC

The request/response communication is a basis for the *remote procedure call* (RPC) model.

- Think of a server as a module (data + methods).
- Think of a request message as a *call* to a server method.
  - Each request carries an identifier for the desired method; the rest of the message contains the arguments.
- Think of the reply message as a *return* from a server method.
  - Each reply carries an identifier for the matching call; the rest of the message contains the result.

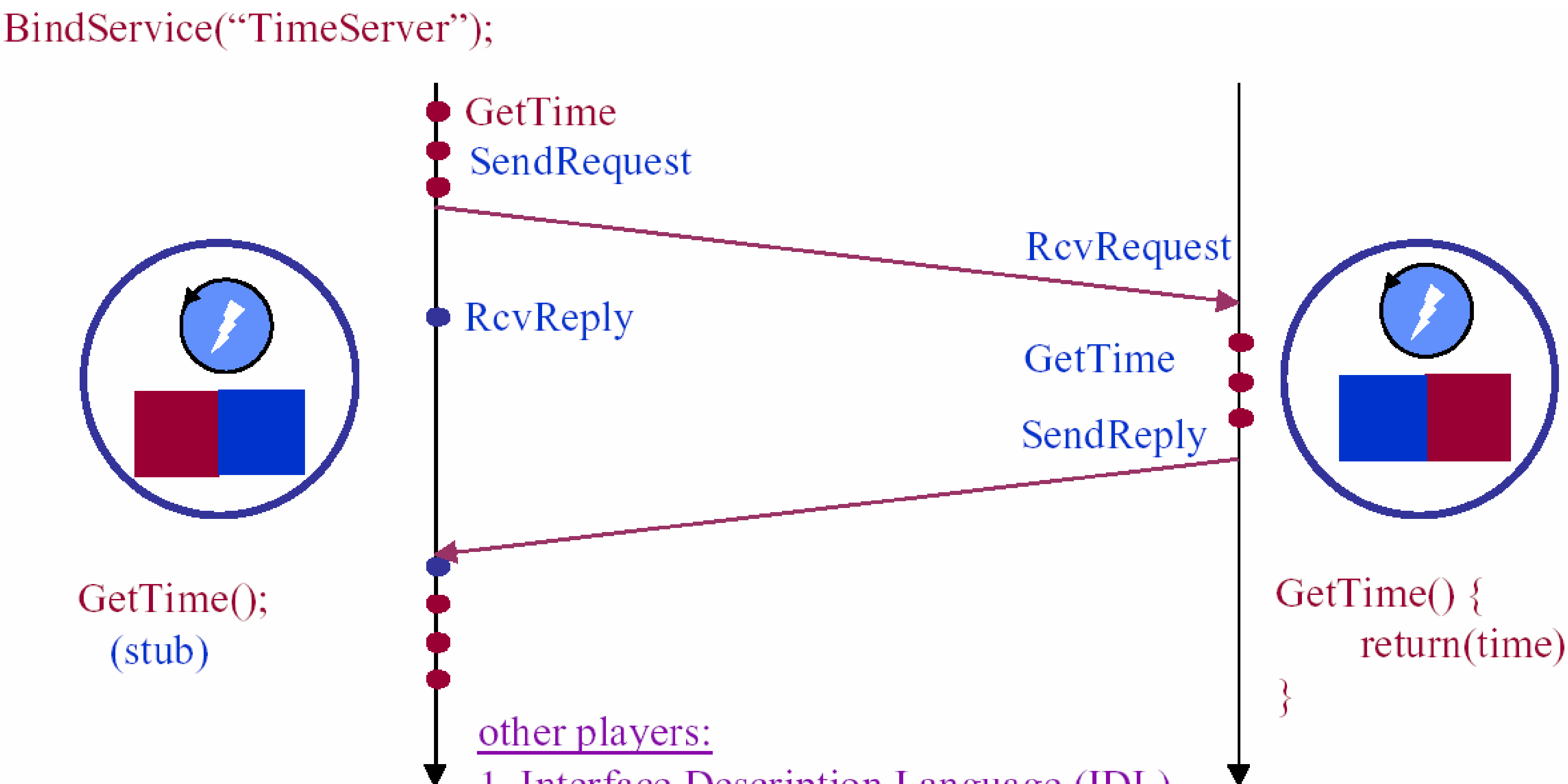
With a little extra glue, the messaging communication can be be hidden and made to look "just like a procedure call" to both the client and the server.

Example: Time Service Using RPC



RPC *stubs* are library routines that handle the details of interacting with the server/client. They may be generated by the system automatically from an abstract description of the service (e.g., a module header file).

Remote Procedure Call Illustrated Remote



other players:

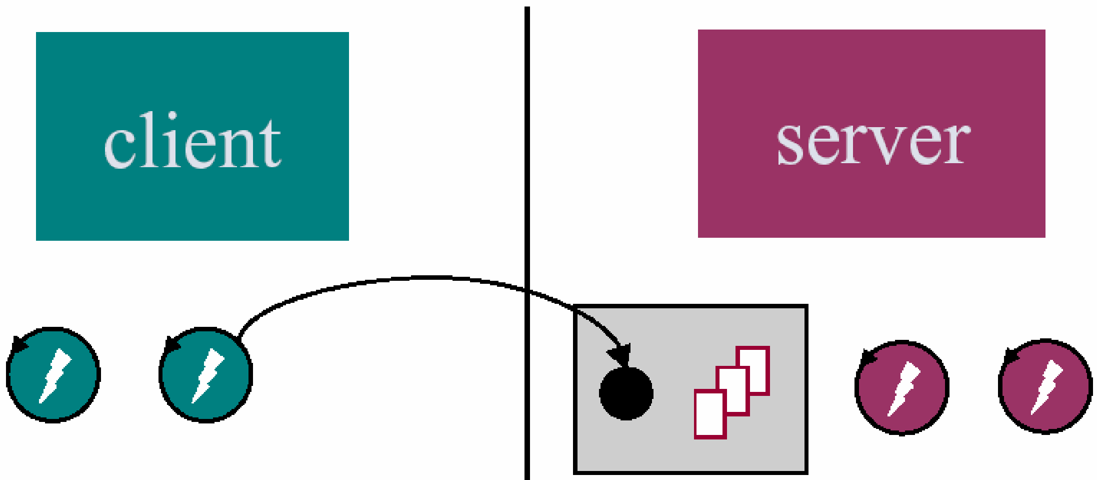
1. Interface Description Language (IDL)
2. stub compiler generates stubs from IDL description of service
3. name registry locates the Time service and returns a *binding*
4. need an argument/result standard for messages (XDR or IIOP)

Some Points About RPC

1. RPC is a syntactically friendly communication/interaction model built *above* basic messaging or other IPC primitives.
   - RPC is a nice model, but it is constrained and not fully transparent; not everyone likes it, and it more-or-less assumes threads.
2. Complex systems may be structured in the usual way as interacting modules, with processes imposing *protection boundaries* crossed using RPC.
   - Interacting processes/modules may fail independently (?).
3. The RPC paradigm extends easily to distributed systems, but a variety of optimizations may be employed in the local cases.
   - e.g., research systems and NT's *LPC* pass arguments in shared memory
4. The RPC model also extends naturally to object-based systems and object-based distributed systems.
   - e.g., research systems, CORBA, Java ***R**emote **M**ethod **I**nvocation*…there is an entire subculture out there

Naming Message/RPC Endpoints: Ports

It may be useful for a given process to manage multiple communication endpoints - often called *ports* - with messages sent to ports rather than processes.




```
Port* svc;
....
svc->send(request);
response = currentThread->receive();
...
```

```
while(systemActive) {
    svc->receive(request);
    ...
    requester->send(response);
}
```

Advantages of Ports

1. Ports decouple IPC endpoints from processes and threads.
    - A thread may send to a port without knowing the identity of the process/thread that receives on that port.
    - Different threads may listen/service the same port, possibly at different times.
2. A thread may listen to multiple ports, separating the message streams designated for different ports.
    - E.g., assign different ports to different objects or virtual services.
3. Ports are a convenient granularity to control message flow.
    - E.g., Selectively enable/disable ports independently, or assign different priorities or access control to different ports.

Some Issues for Port Communication

Issues to consider to design/understand a system with ports:

1. *Asynchrony and notification*. How does a thread know when a message arrives on a port?
   - How to receive from multiple ports, without blocking on an idle port while incoming messages are queued on another?
2. *Naming and binding*. How do threads name the ports to send to or receive from (listen)?
   - How do threads find the names, e.g., for services they want to use?
3. *Protection and access control.*
   - How does the system know if a thread/process has a "right" to send to or listen on a particular port? E.g., how can we prevent untrusted programs from masquerading as a legitimate service?

Examples of Ports in Real Systems

1. Unix sockets and TCP/IP communication.
   - Common primitives/protocols for local messaging and network communication.
   - TCP/IP defines a fixed space of port numbers per node.

     System calls to send/listen to a particular port.
   - Some ports are reserved to processes running with superuser (root) privilege.

     Standard servers in */etc/services* listen at well-known protected ports.
2. Mach supplies a rich set of port/messaging primitives.
   - Open ports (*port rights*) are kernel object handles.
   - Port rights may be passed in messages among processes.

     The only way to get a send/receive right is for some other process to pass it to you! This is a system-wide basis for protection.

The Notification Problem

Communication-oriented systems face an important problem:

How does a client or server know what to do next?

- Servers in networks or server-structured systems might service many clients, possibly on different ports.
- The server must handle messages as they arrive, without blocking to *receive* on an empty port while others have pending messages.

Option 1: Use blocking primitives with lots of threads.

- Leave the scheduling to the thread scheduler.

Option 2: Introduce nonblocking primitives or provide notifications or combined queueing of incoming messages.

- A wide variety of mechanisms have been used: nonblocking polling, Unix *select*, Mach port groups, event queues, etc.

Multithreading: Pros and Cons

Multithreaded structure has many advantages...

- Express different activities cleanly as independent thread bodies, with appropriate priorities.
- Activities succeed or fail independently.
- It is easy to wait/sleep without affecting other activities: e.g., I/O operations may be blocking.
- Extends easily to multiprocessors.

...but it also has some disadvantages.

- Requires support for threads or processes.
- Requires more careful synchronization.
- Imposes context-switching overhead.
- May consume lots of space for stacks of blocked threads.

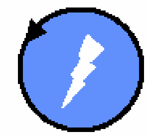
General Alternative: Event-Driven Systems

Structure the code as a single thread that responds to a series of events, each of which carries enough state to determine what is needed and "pick up where we left off".

If handling an event requires waiting for I/O to complete, the thread arranges for another event to notify it of completion, and keeps going, e.g., asynchronous *non-blocking I/O.*

Question: in what order should events be delivered?

```
while (TRUE) {
        event = GetNextEvent();
        switch (event) {
                case IncomingPacket:
                        HandlePacket();
                        break;
                case DiskCompletion:
                        HandleDiskCompletion();
                        break;
                case TimerExpired:
                        RunPeriodicTasks();
                etc. etc. etc.
        }
```

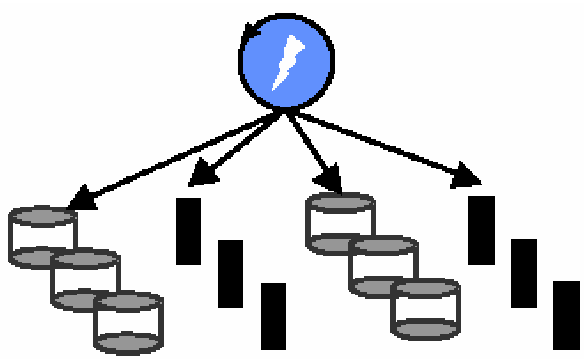


Example: Unix *Select* Syscall

A thread/process with multiple network connections or open files can initiate nonblocking I/O on all of them.

The Unix *select* system call supports such a polling model:

- pass a bitmask for which descriptors to query for readiness
- returns a bitmask of descriptors ready for reading/writing
- reads and/or writes on these descriptors will not block



Select has fundamental scaling limitations in storing, passing, and traversing the bitmaps.

Event Notification with Upcalls

Problem: what if an event needs more "immediate" notification?

- What if a high-priority event occurs while we are executing the handler for a low-priority event?
- What about exceptions relating to the handling of an event?

We need some way to preemptively "break in" to the execution of a thread and notify it of events.

- *upcalls*
- *example*: NT Asynchronous Procedure Calls (APCs)
- *example*: Unix *signals*

Preemptive event handling raises synchronization issues similar to interrupt handling.

Retrospective on IPC

1. There is a continuum of IPC abstractions that combine coordination and data transfer.

   pipes, messages, RPC, RMI

2. IPC may be supported by the kernel interface for processes running on the same machine.
3. IPC abstractions extend easily to networked environments.
4. IPC enables construction of complex software systems from logically independent components.
   - Processes may provide services to other processes; structure applications or the OS itself as a collection of cooperating processes.
   - Trust and access control issues become important.

Summary

1. Threads are a useful tool for structuring complex systems.
   - Separate the code to handle concurrent activities that are logically separate, with easy handling of priority.
   - Interaction primitives integrate synchronization, data transfer, and possibly priority inheritance.
2. Many systems include an event handling mechanism.
   - Useful in conjuction with threads, or may be viewed as an alternative to threads structuring concurrent systems.
   - Examples: Unix signals, NT APCs, *GetNextEvent()*
3. Event-structured systems may require less direct handling of concurrency.
   - But must synchronize with handlers if they are preemptive.