CSC 546: Client/Server Fundamentals

Fall 2000

client/server communications

- overview of underlying concepts and issues
- message passing: naming, address resolution, process synchronization
- RPCs: interaction modes, failure modes, security, data conversion
- message queueing

Message passing

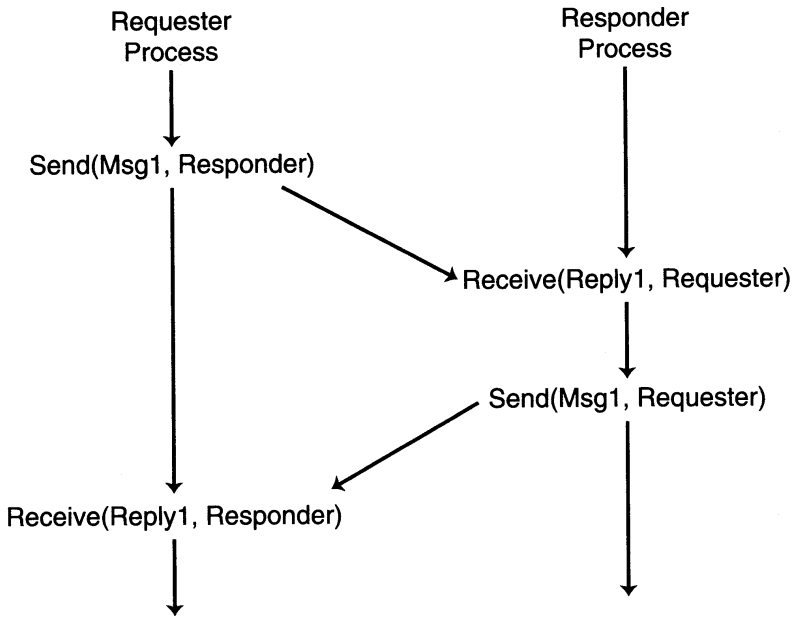
concurrent processes can communicate via:

- shared memory (e.g., semaphores, conditional critical regions)
- message passing through a shared communications channel
  i.e., *interprocess communication (IPC)*

in general, need IPC system calls of the form:

```
send(message, destination)

receive(message, source)
```



Messages and ports

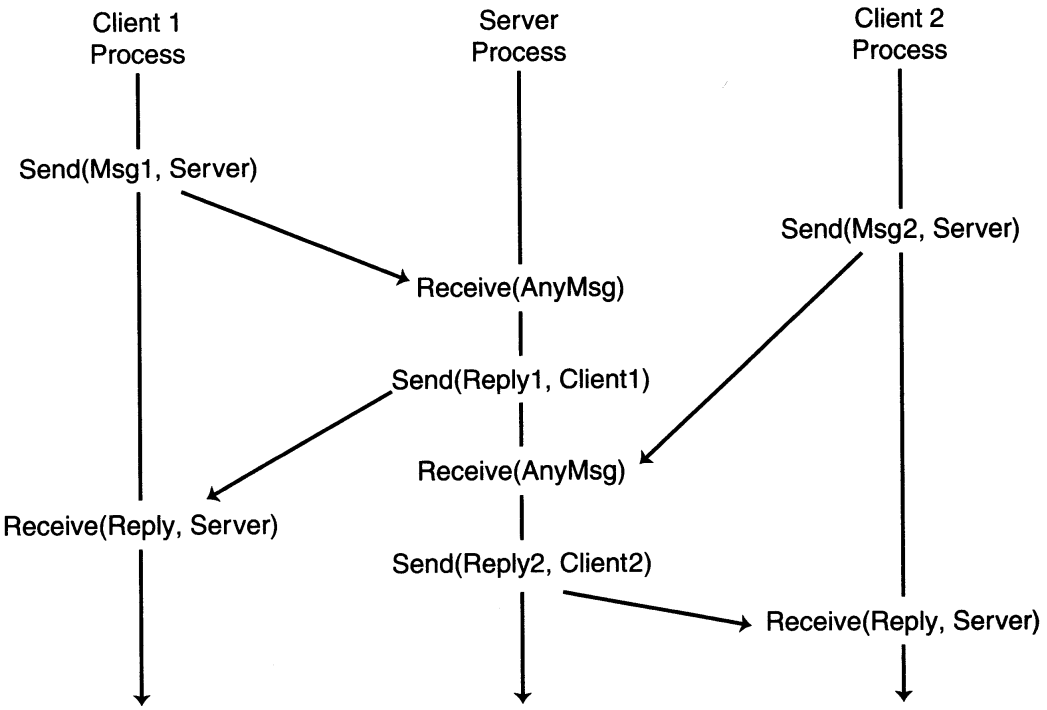
recall: client/server is generally a many/few relationship

- server needs to be able to receive messages from many different clients

a *port* is a "mailbox" assigned to a server process

- any process (client) can send messages to the port
- only one process (server) can receive from the port

messages are received FIFO



Message-passing techniques

3 major issues in message-passing communications

1. type of connection between senders and receivers
2. how messages are addressed
3. how sending and receiving activities are synchronized

Communications can be…

connectionless (a.k.a. packet-switched)

- messages (a.k.a. datagrams) are sent independently,
  each must find its own way to destination (poss. different routes)
- *analogy: courier service*

connection-oriented (a.k.a. circuit-switched)

- communication circuit is established between sender and receiver,
  messages (a.k.a. datastreams) are routed along this dedicated circuit
- *analogy: phone conversation*

tradeoffs:

- connectionless is inherently unreliable (messages can get lost, shuffled)
  can compensate with higher-level protocols
- connection-oriented is inherently reliable (can confirm delivery over circuit)
  overhead in establishing & releasing circuit, confirming receipt

Almost all major protocols support both types of connection

2-way vs. 1-way

a connection-oriented circuit can be:

- duplex: 2-way circuit in which both sides can send simultaneously
  analogy: phone conversation

- half-duplex: 1-way circuit in which only one can send at a time
  requires cooperation to determine who is sending at a given time
  analogy: walkie-talkie

- simplex: 1-way circuit in which transmissions go in only one direction
  not useful for client/server
  analogy: radio transmission

Addressing issues

many ways to designate who you want to communicate with

- by name (object X)
- by address (object at location X)
- by route (object at end of path X)

…

name resolution: X must be bound to the real destination

simplest approach: *static binding*

- specify `(host address, process ID)` pair
- requires client to know physical network address of server
  not practical for client/server: client usually does not know server address

Global namespace

alternatively, have clients & server agree upon a global name

- name must be dynamically bound to server's address at run time

names must be unique within their context

e.g., on a single machine, process ID suffices
in LAN, need host name/address + process ID

- hierarchical concatenation: append names as context widens
  e.g., `network + subnetwork + host + process ID + name`

- range partitioning: use uniform naming format and partition values
  e.g., host 1 uses 0000-0999, host 2 uses 1000-1999, …

- hybrid approach: partition values along a hierarchy
  e.g., Internet Domain Naming Scheme (DNS)
  `davereed@bluejay.creighton.edu`

Dynamic binding

via convention

e.g., configuration file containing global names and network addresses
or, file-sharing of configuration file over a network
requires updating the configuration file as machines are added/removed

via broadcast

e.g., NetBIOS protocol, uses memory-resident system table

1) client requests connection with server via global name
2) NetBIOS checks to see if name/address is stored in table
3) if not, broadcasts name to the entire network
4) server with that name responds with its network address
5) NetBIOS updates system table

via name server

e.g., separate server responds to requests for network addresses
but how does the client find the name server???

Synchronization issues

message-passing protocols can be:

- blocking
  sending process is suspended until message is sent & acknowledged
  synchronous: server can know state of client when receives message
  easiest to program

- non-blocking
  sending process continues execution after sending the message
  asynchronous: client state may have changed by time server receives
  used in real-time systems where sender can't wait for response

All major IPC protocols provide both synchronous & asynchronous

Remote Procedure Calls (RPCs)

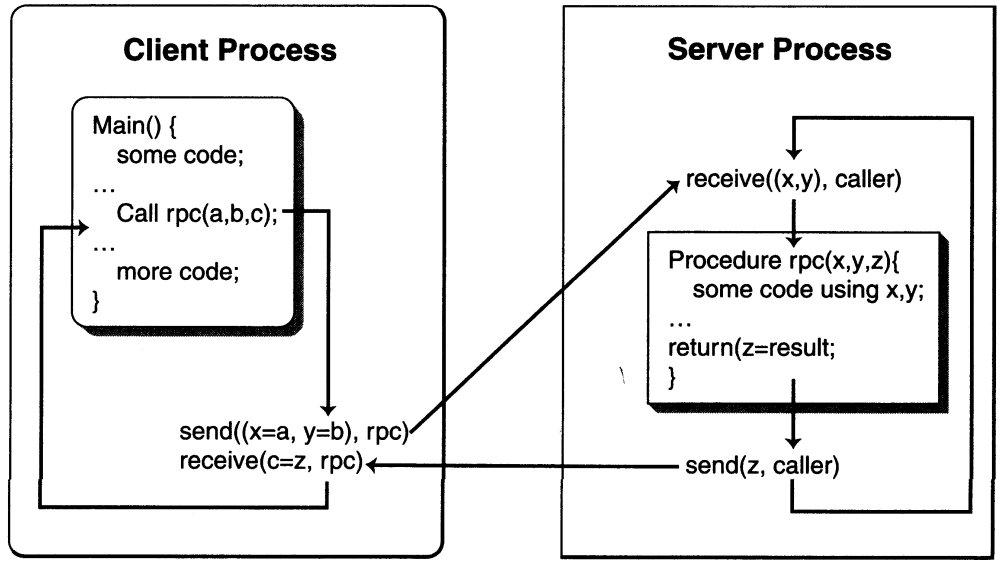
managing concurrent processes is difficult

- timeouts, race conditions, retries, garbled messages, …

client/server communication is made much simpler by generalizing the procedure call model to concurrent processes

1. client makes remote procedure call
2. calling process is suspended, parameters are sent to the server, and procedure is executed there
3. when procedure completes, results are sent back to the client
4. client resumes processing

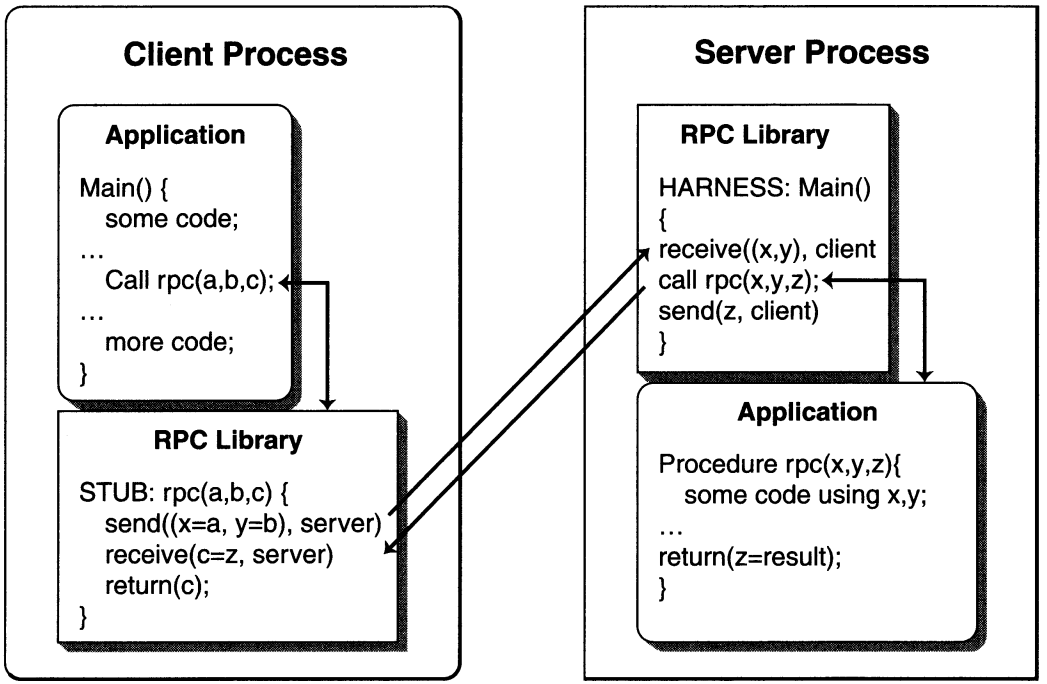
Note: if global variables are allowed, then the RPC call must send process context to the server



RPCs (cont.)

from the application programmer's perspective,

must link to client's RPC libraries, then to server's RPC libraries



Note: RPC's are inherently synchronous

- send/receive messages must utilize blocking

Implementing RPCs

interaction models:

- most networks require the server to be running when RPC is requested

  e.g., loop model : server runs in continuous loop, blocked on a `receive` when RPC request arrives, server wakes up and processes

  to improve performance, server usually handles each request by spawning a slave process for handling it

  e.g., rendezvous model: server can do other work between requests, checks for requests at specific intervals

- RPC communication is usually thought of as connectionless

  if numerous calls are to be made, a connection-oriented model can be used

  most RPC protocols support both styles of communication

Implementing RPCs (cont.)

failure modes

- if client crashes after RPC request
  procedure call is wasted

- if RPC request or results are lost in transmission
  must use timeouts to keep client from waiting forever
  (resends request if no response after X time)

- if server crashes while executing RPC
  client will timeout and resend request (but not forever!)

An RPC protocol should either:

- treat transactions as atomic (roll back RPC if not acknowledged), OR
- require that applications perform all recovery activity

Implementing RPCs (cont.)

security

- RPC protocol layer sits between application layer and transport layer
  security is dependent upon the capabilities provided from below

- ideally, a secure RPC protocol would provide ways to:
  1. identify the sender of a message
  2. verify the sender's identity
  3. control how the message is routed
  4. detect any message tampering
  5. encrypt the message

- common technique for detecting tampering: cyclic redundancy check
  same basic idea as a parity bit: add message-dependent data to end,
  if message has been changed, then parity data will no longer match

Implementing RPCs (cont.)

data conversion

- different machines represent data differently, organize memory differently

  e.g., ASCII vs. EBCDIC codes

  ASCII(77) = 'M' EBCDIC(77) = '('

  e.g., little-endian (Intel) vs. big-endian (Sun SPARC)

  little-endian stores words with most significant byte first

  big-endiant stores words with least significant byte first

- RPC protocol must provide conversion services, or require common format

performance

- RPCs are much more expensive than calling local procedures
- requires 2 IPC calls (receive & send), network latency, context switching,…

local procedure call	10 µsec	RPC to remote process	6,000 µsec
IPC to local process	300 µsec	local disk I/O	12,000 µsec

Message queueing

can simplify communications using a message queue

- application creates explicit queues
- client requests are put onto queues, picked off by server
- server responses can be queued as well

message queues serve to decouple the sender and receiver

message queue is an application-level version of a port

- both involve sending/receiving to an abstract address
- message queue is visible to application, can be controlled
- port is a low-level, hidden to applications (but may be used to implement)

Note: message queues are inherently asynchronous

When message queueing?

connectivity is intermittent or costly

- e.g., mobile computing

message passing can be scheduled

- e.g., nightly report requests

multiple, identical servers exist to process requests

- e.g., news servers

message arrivals are unpredictable, can be overwhelming

- e.g., online auction

verification of message source/receipt is desired

- e.g., credit card transaction

messages need to be prioritized (not just FIFO)

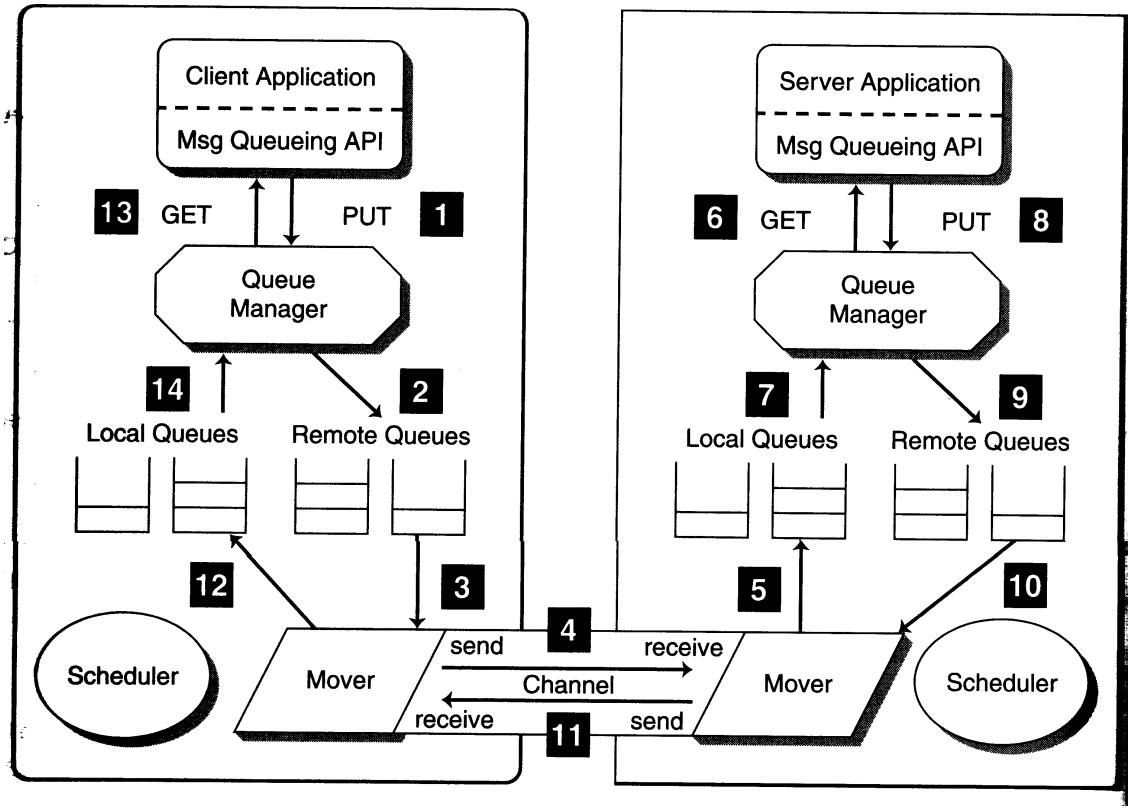
- print scheduler

Queue management

queue manager serves as interface between application and queues

- accepts and authenticates application requests
- stores and retrieves requests from local queues
- administers local queues

1. app issues PUT request
2. manager places request on outbound queue
3. periodically, mover process is instructed to send requests
4. client mover sends requests over channel to server
5. mover process places requests in inbound queue
6. server issues GET request when ready
7. manager retrieves request from inbound queue

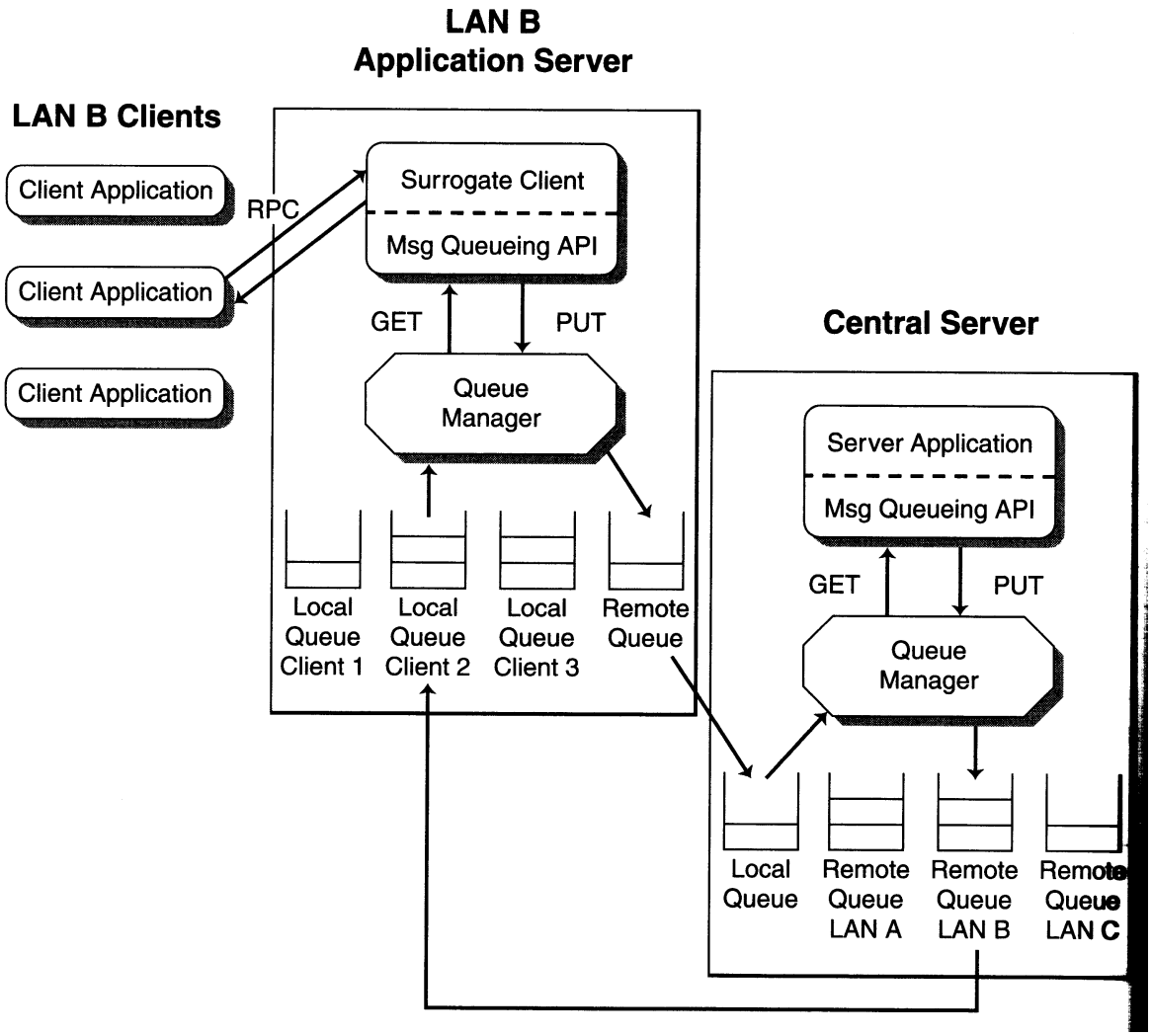


RPC + queueing

can combine RPCs and message queueing in same application

common use:

- clients use RPCs to communicate with application server (e.g., front-end)
- application server uses message queueing to communicate with central server (e.g., DBASE)



Next week…

The Web as a client/server system

- HTTP
- caching
- server-side programming
- cookies

Read online materials (& do some digging on your own)

As always, be prepared for a short quiz