Interprocess Communication (IPC) and Coordination

Topic 3

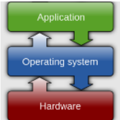
Hartmut Kaiser

https://teaching.hkaiser.org/fall2025/csc7103/

Causality

Causality

- The lack of a global system state – fundamental property of a distributed system
  - Most of the time distributed systems are asynchronous
- Distributed systems are causal – the cause precedes the effect
  - The sending of a message precedes the receipt of a message
  - The distributed system is composed of the set of processors and there are multiple sets of events that occur on these processors
    - Events include message send, message receipt, user input receipt, signal raising, output creation, etc.
  - How to define the ordering among different events:
    - We write $e_1 < e_2$ if we know that event e_1 occurred before event e_2
- In distributed systems, it is difficult to deduce which event came first
  - Need to combine information from different sources to determine the ordering. If information source I tells us that e_1 occurred before e_2, we write $e_1 <_I e_2$



Causality Definitions

- Event e_1 causally happened before event e_2 (that is, $e_1 <_H e_2$):
- Transitive closure of the processor orderings and the message orderings
  - Processor ordering: e_1 occurred before e_2 in the same process/processor p

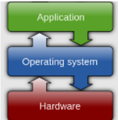
$$e_1 <_P e_2$$

    - Events that occur on the same processor are totally ordered
  - Message ordering: A message (m) sent by the process p_i after e_1 occurred is received by the process p_j before e_2 occurred

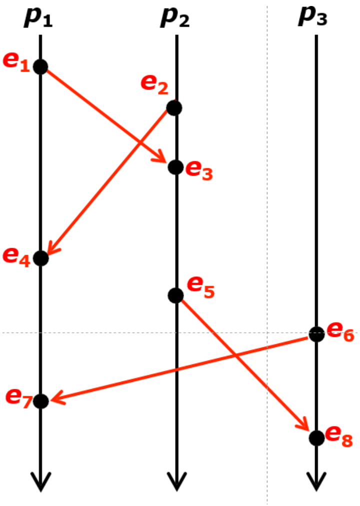
$$e_1 <_m e_2$$

    - Simply, e_1 is the sending of message m and e_2 is the receipt of message m
  - Transitive closure property: if e_1 causally happened before e_2 and e_2 causally happened before e_3, then

$$e1 <_C e3$$



Happens-Before DAG



Causally ordered events:

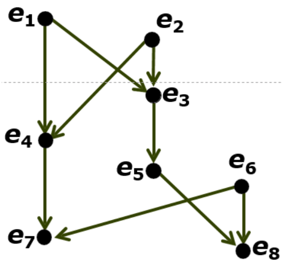
$e_1 <_{P1} e_4 <_{P1} e_7$

$e_1 <_m e_3$

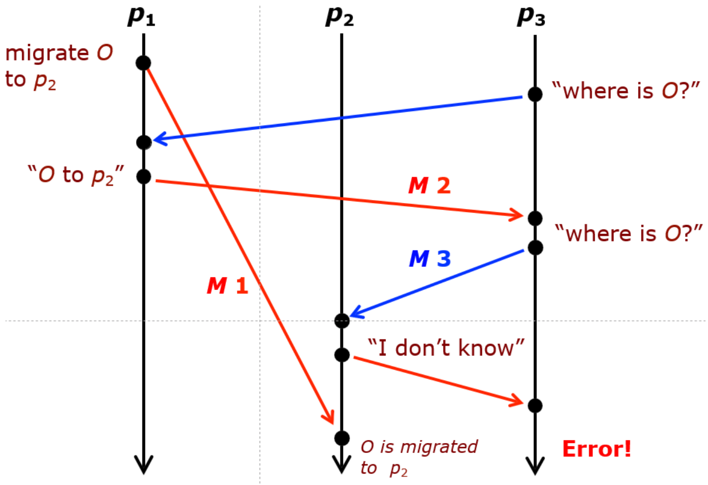
Concurrent (disjoint) events

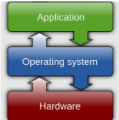
e_1 **and** ***e_6***

DAG (directed acyclic graph)



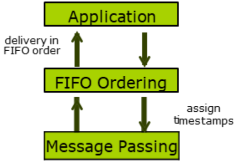
Example: Causality Violation

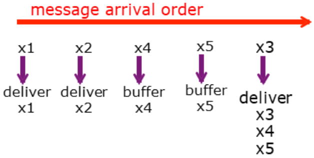


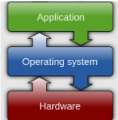


Causality Communication

- Ensuring that processor never experiences a causal violation
- Protocol for causal communication:
  - A processor cannot choose the order in which messages arrive but it can change the order in which messages are delivered to the applications that consume them
  - Revise delivery order by holding back messages that arrived "too soon"
  - The source attaches timestamps on messages (to order messages), and the destination delays the delivery of out-of-order messages
- Protocol for FIFO message delivery (TCP communication)



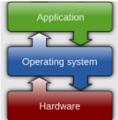




IPC and Synchronization

Language Mechanisms for Synchronization

- A concurrent programming language supports:
  - Specification of concurrent processing
  - Synchronization of processes
  - Interprocess communication
  - Non-deterministic execution of processes
- How the normal OS approaches can be extended to the distributed OS
- Various synchronization mechanisms
  - Shared-variable approaches: Semaphore, monitor, conditional critical region, serializer, path expression
  - Message passing approaches: Communicating sequential processes, remote procedure call, rendezvous
- Classic synchronization example: Concurrent readers/exclusive writer problem

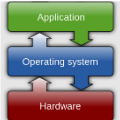


Critical Section Problem

- Multiple processes are competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed
- Problem – ensure that when one process is executing in its critical section, no other processes are executing in their critical sections
  - Mutual exclusion should be enforced
- Entry section implements a process' request to enter its critical section which is followed by an exit section
  - Processes may share some common variables to synchronize their actions (to have orderly execution of cooperating processes)

General structure of process p_i

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```



Semaphores

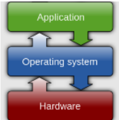
- Semaphore is a synchronization tool
  - Works like mutex locks to enforce mutual exclusion

- Semaphore S – protected integer variable which can only be accessed via two operations

```
wait(S){
      while (S≤ 0)
         ; // no operation (busy wait)
         S--;
}
signal (S){
      S++;
}
```

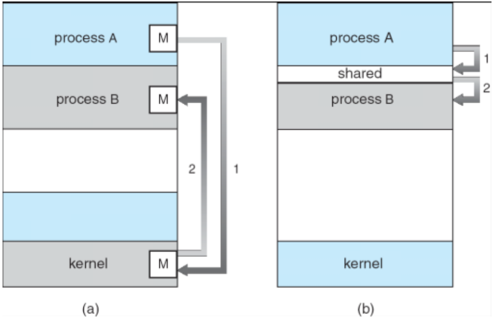
Original terminology:
P (from Dutch proberen, to test) for **wait**
V (from verhogen, to increment) for **signal**

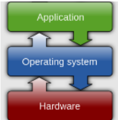
- These operations are indivisible (atomic), that is, only one process can modify the semaphore value at a time



Interprocess Communication Models

- Message passing – Useful for exchanging smaller amounts of data; easier to implement through system calls but slower
- Shared memory – Allows maximum speed and convenience of communication; faster accesses to shared memory





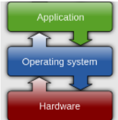
Message-Passing Synchronization

- A mechanism for cooperating processes to communicate and to synchronize their actions without sharing the same address space
  - The only means of communication in distributed systems without shared memory
- Message-passing facility provides two operations:

  `send(message)` – message size fixed or variable

  `receive(message)`
- Two processes wishing to communicate need to establish a communication link between them and exchange messages via

  `send/receive`
  - Implementation of communication link physical (e.g., shared memory, hardware bus or network) or logical
- Message can be asynchronous or synchronous



Asynchronous Message Passing

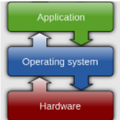
- Assumes non-blocking send and blocking receive – uses the channel with an unbounded buffer as a semaphore (message content is not important)
- Can be useful as semaphore if communication channel can be specified
  - Blocking receive – (acquiring the lock), Non-blocking send – (releasing the lock)
- Mutual exclusion solution using asynchronous message passing:

process p_i

```
begin
    receive(channel)
    critical section
    send(channel)
end
```

channel server

```
begin
    create channel
    send(channel)
    manage channel
end
```



Synchronous Message Passing

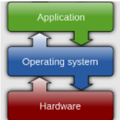
- Assumes blocking send and blocking receive – symmetrical waiting
- Rendezvous between send and receive
  - Allows two processes to join and exchange data at a synchronization point and continue their separate execution thereafter
- Mutual exclusion solution using synchronous message passing:

process p_i

```
begin
        send(sem, msg)
        critical section
        receive(sem, msg)
end
```

semaphore server

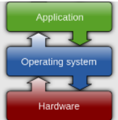
```
loop
        receive(pid, msg)
        send(pid, msg)
end
```



Communication and Coordination

Communication and Coordination

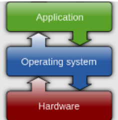
- Cooperating processes must interact with each other using some forms of communication model to coordinate their execution
- Interprocess communication (IPC): Two approaches are message passing and shared memory
  - Message passing - only method of exchanging data/information between processes in distributed systems
  - All higher level models must be built on the top of message passing
- Request/reply – based on the client/server concept
- Transactions – sequences of request/reply communications that require communication atomicity
  - Only logically shared memory (data objects) simulated by message passing is possible in distributed systems
- Name service model: Locating the communication entities (objects)
- Distributed process coordination:
  - Classical problems: Distributed mutual exclusion and leader selection



Different Levels of Communication

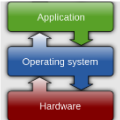
- Five levels of communication abstraction
  - Top three levels deal with the transfer of messages among distributed processes

Interprocess Communication	Transaction
	Request/Reply (RPC)
	Message Passing
Networking Operating Systems	Transport Connection
Communication Network	Packet Switching



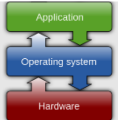
Message Passing Communication

- Communicating processes pass composed messages to the system transport service, which provides connectivity for message transfer in the network
  - Basic communication primitives
  - Message synchronization and buffering
  - Pipe and socket APIs
  - Group communication and multicast



Basic Message Passing Primitives

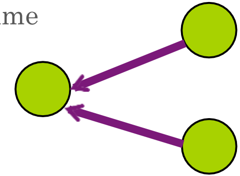
- Two generic message passing primitives
  - Send (destination, message)
  - Receive (source, message)
- The communication entities, source and destination, can be addressed in four different ways
  - Process name
  - Link
  - Mailbox
  - Ports
- Message size can be fixed or variable



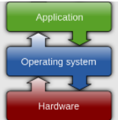
Source/Destination Identification

- Process names or unique global process identifiers are required
  - May be obtained by adding machine address to process id
  - Symmetric/asymmetric addressing options
    - Symmetric – sender and receiver need to explicitly name each other
    - Asymetric – only sender needs to indicate the receiver
- Allows one logical communication path/link between a pair of sending and receiving processes
- Process identifiers need to be known at coding time

symmetric process name



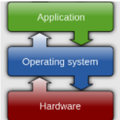
asymmetric process name



Links

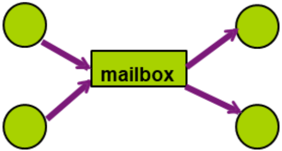
- Identifying/specifying each path in the communication primitives as connection or link (similar to virtual circuit concept)
  - Allows multiple data paths between processes
  - Different links, each pointing to an actual communication path can be used
- Direct communication between peer processes can be provided by using process names and link numbers

Two links using two different link numbers

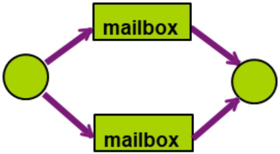


Mailboxes

- Mailboxes are global data structures shared by some sender and some receiver processes
  - Messages are sent to and received from mailboxes
- Allow indirect communication between sender and receiver processes
- Allow multipoint and multipath communication



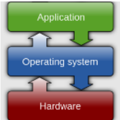
multipoint communication



multipath communication

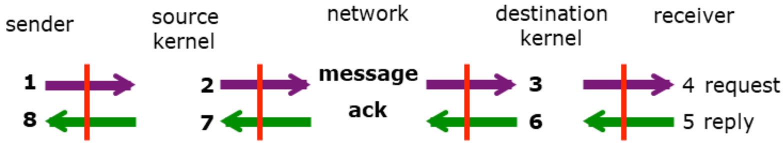
Ports

- Port is an abstraction of a finite-size FIFO queue maintained by the kernel
  - A special example of mailbox
  - Messages can be appended to or removed from the queue by send and receive operations
  - Ports are bidirectional and buffered, and support indirect communication
- Created by user processes using system calls
  - Referenced by port numbers
  - User ports are mapped to transport ports and vice versa

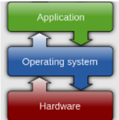


Message Synchronization

- Message passing communication depends on synchronization at several points
  - Between user process and system kernel
  - Between kernel and kernel
  - Between source and destination processes
- Send/receive primitives may be blocking or non-blocking
  - Blocking primitive means that the calling process needs to be blocked for the message delivery or receipt

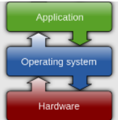


Message synchronization stages



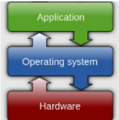
Buffering

- Common default: a non-blocking send and a blocking receive
  - Non-blocking send also referred to as an asynchronous send
- Blocking send may be of different types:
  - Ordinary blocking send
  - Reliable blocking send
  - Explicit blocking send
  - Request and reply – called client/server communication
- Blocking receive implies that the process can not continue till the message is received
- Buffering is crucial in the synchronization:
  - The sender puts messages in the buffer while the receiver removes the message from the buffer
  - Sharable buffer spaces smooth out the asynchronous processing of messages
  - One big buffer by combining the buffers in the sender kernel, the receiver kernel and the communication network



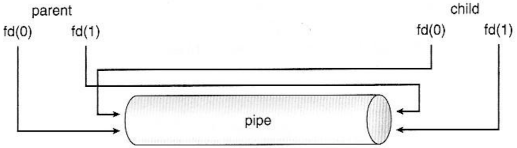
Application Program Interface

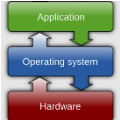
- User processes communicate using an API, independent of the underlying communication platform
  - Shared communication channels are (logically) shared objects
  - Internal details and implementation managed by the kernel are transparent to the users
- Used in both Windows and Unix environments
- Pipes and socket APIs



Pipes

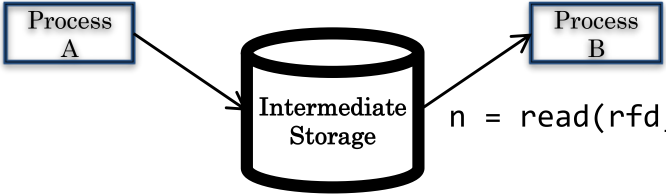
- Pipes are implemented with a finite-size, FIFO-byte stream buffer maintained by the kernel
  - A pipe serves as an unidirectional communication link
  - A `pipe` system call returns two pipe descriptors, one for reading and the other for writing: `fd(0)`: read-end, `fd(1)`: write end
- Ordinary pipes: used only for related processes (pipe descriptors are shared by parent process and children)
- Named pipes: FIFO files shared by unrelated (disjoint) processes across different machines with a common file system - limited to a single domain





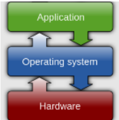
Communication Between Processes

```
write(wfd, wbuf, wlen);
```



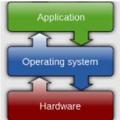

```
n = read(rfd, rbuf, rmax);
```

- Data written by A is held in memory until B reads it
- Queue has a fixed capacity
  - Writing to the queue blocks if the queue if full
  - Reading from the queue blocks if the queue is empty
- POSIX provides this abstraction in the form of pipes



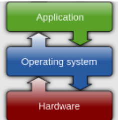
Pipes

- `int pipe(int fileds[2]);`
  - Allocates two new file descriptors in the process
  - Writes to `fileds[1]` read from `fileds[0]`
  - Implemented as a fixed-size queue



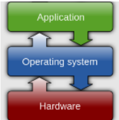
Single-Process Pipe Example

```
#include <unistd.h>
int main(int argc, char *argv[]) {
  char *msg = "Message in a pipe.\n";
  char buf[BUFSIZE] = { '\0' };
  int pipe_fd[2];
  if (pipe(pipe_fd) == -1) {
    fprintf (stderr, "Pipe creation failed.\n"); return EXIT_FAILURE;
  }

  ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
  printf("Sent: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

  ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
  printf("Rcvd: %s [%ld]\n", buf, readlen);

  close(pipe_fd[1]); close(pipe_fd[0]);
}
```



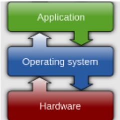
Inter-Process Communication (IPC)

```c
pid_t pid = fork();
if (pid < 0) {
  fprintf (stderr, "Fork failed.\n");
  return EXIT_FAILURE;
}
if (pid != 0) {
  ssize_t writelen = write(pipe_fd[1], msg, msglen);
  printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
  close(pipe_fd[0]);
  close(pipe_fd[1]);
} else {
  ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
  printf("Child Rcvd: %s [%ld]\n", msg, readlen);
  close(pipe_fd[0]);
  close(pipe_fd[1]);
}
```



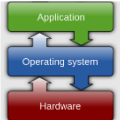
Named Pipes

```c
#include <unistd.h>

// create named pipe
if (mkfifo("/tmp/my_fifo", S_IRUSR|S_IWUSR) == -1) {
    perror("mkfifo"); return 1;
}

// delete the named pipe
if (unlink("/tmp/my_fifo") == -1) {
    perror("unlink"); return 1;
}
```



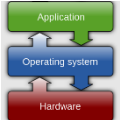
Named Pipes

```c
// write to named pipe
int fd = open("/tmp/my_fifo", O_WRONLY);
if (fd == -1) {
    perror("open"); return 1;
}

char *message = "Hello, Named Pipe!";
if (write(fd, message, strlen(message) + 1) == -1) {
    perror("write"); return 1;
}

close(fd);
```



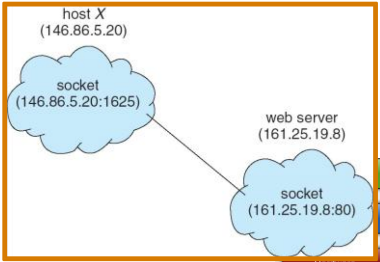
Named Pipes

```c
// read from a named pipe
int fd = open("/tmp/my_fifo", O_RDONLY);
if (fd == -1) {
    perror("open"); return 1;
}

char buffer[100];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
if (bytes_read == -1) {
    perror("read"); return 1;
}

buffer[bytes_read] = '\0';
printf("Received message: %s\n", buffer);

close(fd);
```



Sockets

- Sockets provide two-way communication links shared by processes across heterogeneous domains
- A socket is an endpoint for a communication link managed by the transport service
  - A pair of processes communicating over a network employs a pair of sockets – one for each process
  - Socket system call returns a socket descriptor (logical communication endpoint (local to a process), which must be associated with a physical communication endpoint – bind system call
- A physical communication endpoint is specified by a network host address and transport port pair
  - Each socket is made up of an IP address concatenated with a port number:
  - The socket 146.86.5.20:1625 refers to port 1625 on host 146.86.5.20



Sockets in Schematic

Client

Create Client Socket

Connect it to server (host:port)

Connection Socket ⟺ *Connection Socket*

write request

read response

Close Client Socket

Server

Create Server Socket

Bind it to an Address (host:port)

Listen for Connection

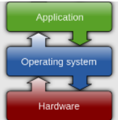
Accept syscall()

read request

write response

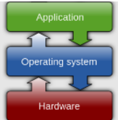
Close Connection Socket

Close Server Socket



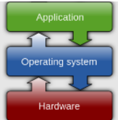
Client Protocol

```
char* host_name = "www.lsu.edu";
char* port = "80";

// Create a socket
struct addrinfo *server = lookup_host(host_name, port);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                     server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

// Clean up on termination
close(sock_fd);
```



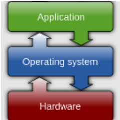
Server Protocol

```
// Create socket to listen for client connections
char *port = "80";
struct addrinfo *server = setup_address(port);
int server_socket = socket(server->ai_family, server->ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {   // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}
close(server_socket);
```



Client: Getting the Server Address

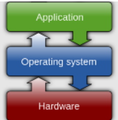
```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  // hints.ai_flags = AI_PASSIVE;

  int rv = getaddrinfo(host_name, port, &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```



Server Address: Itself

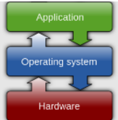
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  getaddrinfo(NULL, port, &hints, &server);
  return server;
}
```

- Accepts any connections on the specified port



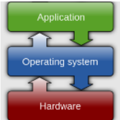
SSL

- Sockets are widely used and need communication security.
- Secure socket layer (SSL) provides – Privacy, Integrity, Authenticity
- Privacy and integrity are maintained by handshake protocol and cryptography
  - Handshake protocol establishes communication session (write) keys and message authentication check, and validates the authenticity of clients and servers
    - The server is verified with a certificate assuring client is talking to correct server
    - Asymmetric cryptography used to establish a secure session key (for symmetric encryption later) for bulk of communication during session
    - Communication between each computer then uses symmetric key cryptography
  - Record layer protocol handles fragmentation, compression/ decompression, encryption/decryption of messages records
- Authentication is done by third-party certification authority

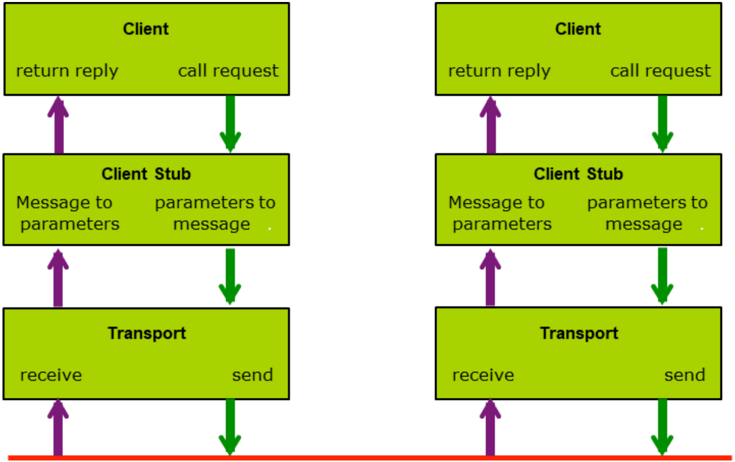


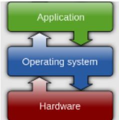
Request/Reply Communication

- Service-oriented request/reply communication is above basic message passing – next level of communication
  - The sender is blocked (or the message is considered not delivered) until it receives a reply
- RPC – remote procedure call
  - Is a language-level abstraction to support request/reply communication mechanism based on message passing
  - Represents a pair of synchronization request (calling a remote procedure) and reply (waiting for results) communications
  - Abstracts procedure calls between processes on networked systems, providing access transparency to remote operations
- RPC is implemented by stub procedures at both the client end and the server end
  - Client-side stub locates the server and marshals the parameters
  - Server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server



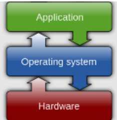
RPC Flow


Client
return reply
call request
Client Stub
Message to parameters
parameters to message
Transport
receive
send
Client
return reply
call request
Client Stub
Message to parameters
parameters to message
Transport
receive
send


Application
Operating system
Hardware

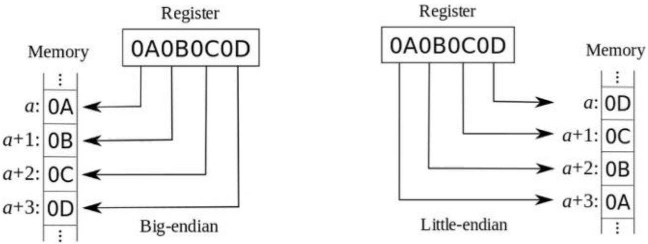
RPC Implementation

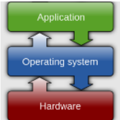
- Parameter passing and data conversion – parameter marshaling
  - Parameters are passed by call-by-value and call-by-copy/restore
  - Data typing, data representation, data transfer syntax problems can be solved using an universal language or canonical data representation
- Binding between the client and the server – match maker
  - Port mapper to provide the port number of the requested server to the client
  - Directory server to locate the server machine if it is unknown
- RPC Compilation - three major components:
  - Interface specification file, RPC generator, run-time library
- RPC exception and failure handling
- Secure RPC



RPC Implementation (Cont’d)

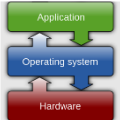
- Data representation handled via External Data Representation (XDL) format to cope with different architectures
  - Big-endian (most significant byte first) and little-endian (least significant byte first)
- Remote communication has more failure scenarios than local
  - Messages can be delivered exactly once rather than at most once
- OS typically provides a rendezvous (or matchmaker) service to connect client and server





RPC Exception and Failures Handling

- Exception handling
  - Overflow/underflow or protection violation in procedure execution
  - In-band or out-band signaling for the exchange of status and control information
- Failure handling
  - Not locating the server, link failure, delayed or lost messages
  - Idempotent services – a request can be repeatedly executed
  - Detecting a duplicate or out-of-sequence request message – the client attaches a sequence number to each request
  - Reliable transport layer (TCP connection)
- Server crash and client crash
  - Generally difficult to deal with
  - Using a time-out or waiting for the failed server/client to come back



RPC Implementation

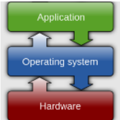
- Interface description language (IDL), here XDR language

```
program KVSTORE {
    version KVSTORE_V1 {
        int EXAMPLE(int) = 1;
    } = 1;
} = 0x20000001;
```

- Use this to generate stubs:

```
rpcgen kv_store.x
```

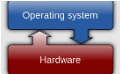
- Generates client and server files



RPC Implementation

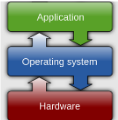
```c
/* Generated client RPC stub. */
int *
example_1(int *argp, CLIENT *clnt)
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, EXAMPLE,
        (xdrproc_t) xdr_int, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

```c
/* User supplied client RPC stub. */
int example(int input) {
  CLIENT *clnt = clnt_connect(HOST);

  int ret; int *result;

  result = example_1(&input, clnt);
  if (result == (int *)NULL) {
    clnt_perror(clnt, "call failed");
    exit(1);
  }
  ret = *result;
  xdr_free((xdrproc_t)xdr_int, (char *)result);

  clnt_destroy(clnt);
  return ret;
}
```



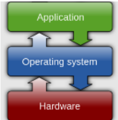
RPC Implementation

```c
/* Example server-side RPC stub. */
int *example_1_svc(int *argp, struct svc_req *rqstp) {
  static int result;

  result = *argp + 1;

  return &result;
}
```



Secure RPC

- Security is important for RPC
  - RPC opens doors for attacks from unfriendly remote users
  - RPC supports all types of client/server computations
- The primary security issues are
  - Authentication of client and server processes
  - Authenticity and confidentiality of messages
  - Access control authorization from client to server
- Authentication protocol for RPC must establish:
  - Mutual authentication for messages and communicating processes
  - Message integrity, confidentiality, and originality
- Designing secure authentication protocol is complex matter
  - Example: Sun's Secure RPC



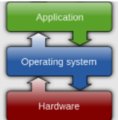
Transaction Communication

- Transactions in communication are a set of asynchronous request/reply communications generally involving the multicast of the same message to replicated servers and different requests to partitioned servers
  - Similar to fundamental unit of interaction between client and server processes in a database system

- Transaction is collection of instructions or operations that performs single logical function
  - A series of read and write operations

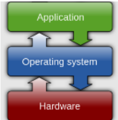
- Example: Consider two data items A and B, and consider two transactions T_0 and T_1
  - Execute T_0, T_1 atomically
  - Execution sequence called schedule
  - Atomically executed transaction order called serial schedule

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)



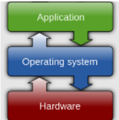
ACID Properties

- Transaction communications must satisfy the ACID properties:
  - Atomicity: all or nothing
  - Consistency/serializability: interleaving results in serial execution in some order
  - Isolation: partial results are not visible outside
  - Durability: after committing, the results will be made permanent
- Ensuring ACID properties requires that the participating processors coordinate their execution of a transaction
  - Challenging in a distributed system because several sites may be participating; any site or link failure may result in erroneous computations
  - Each site has its local transaction coordinator and maintains a log for recovery
  - Name the processor which initiates the transaction the coordinator and name the remaining processors the participants



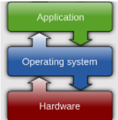
Two-Phase Commit Protocol

- The two-phase commit (2PC) protocol is analogous to a real-life unanimous voting scheme
  - One coordinator and multiple participants for a distributed transaction **T**
  - Each of them have access to some stable storage to maintain its activity log
  - **T** is committed only if all participants agree and ready to commit
- Coordinator (initiator site):
  - Prepare to commit the transaction **T** by writing every update in activity log
  - Write a precommit record in activity log, and multicast a vote request to all participants asking whether they are ready to commit
  - If all participants vote YES within a time-out period, multicast a commit message. Otherwise, multicast an abort message
- Participant (other participating sites):
  - Upon receiving the vote request, prepare to commit the transaction **T** by writing every update in activity log
  - Write a precommit into the log and sends a YES reply to the coordinator. Otherwise, abort **T** and send a NO reply to the coordinator
  - Wait for a commit message from the coordinator. If received, commit **T**. If abort message is received, abort **T**



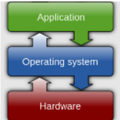
2PC Algorithm for Coordinator

```
2PC_Coordinator()
    precommit the transaction
    For every participant p,
        send(p,VOTE_REQ)
        wait up to t seconds for VOTE messages
            Vote(sender; vote response):
                if vote_response = YES
                        increment the number of yes votes
        If each participant responsed with a YES vote
            commit the transaction
            for every participant p,
                send(p,COMMIT)
        else
            abort the transaction
            for every participant p,
                send(p,ABORT)
```

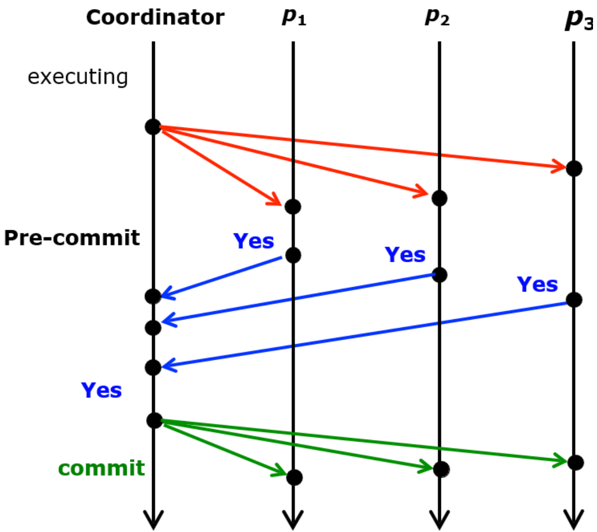


2PC Algorithm for Participant

```
2PC_Participant()
    While True
        wait for a message from the coordinator
        VOTE_REQ(coordinator)
            if I can commit the transaction
                precommit  the  transaction
                write a YES vote to the log
                send(coordinator,YES)
            else
                abort the transaction
                send(coordinator,NO)
        COMMIT(coordinator)
            commit the transaction
        ABORT(coordinator)
            abort the transaction
```



2PC Protocol - Example



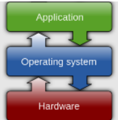
First phase:

Obtain the votes from all participants

Second phase:

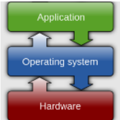
Distribute the agreement to commit

Find the stable property that every processor voted Yes



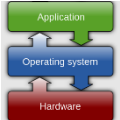
2PC Protocol - Recovery

- When used with an activity log in stable storage, 2PC protocol is highly resilient to processor failures
  - The activity log can be replayed upon the recovery of a failure
  - Note that every participant is required to vote, and once a processor votes it is not allowed to change its vote
- Three types of failure and recovery actions:
  - Failures before a precommit
    - A processor (coordinator or participant) can simply abort the transition
  - Failures after a precommit but before a commit
    - Coordinator can abort the transaction or attempt to commit the transaction by re-multicasting (retake the vote)
    - Participant recovery is complicated: needs to check with the coordinator or other participant about the transaction status
  - Failures after a commit
    - Coordinator resends the commit message to finish the transaction Participant simply makes the transaction's updates permanent



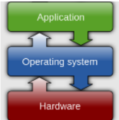
Group and Multicast Communication

- Besides point-to-point communication, multipoint group communication is naturally expected in distributed systems
  - Notion of a group is essential for cooperative software
  - Managing group of processes or objects needs multicast communication
- Issues/complications of multicast communication implementation
  - Reliability: Best effort vs. reliable
  - Failures
  - Delivery order
  - Overlapping groups



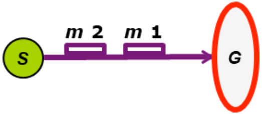
Multicast Issues

- Reliable delivery issue in multicast
  - Two multicast application scenarios: Soliciting a service from any server or requesting a service from all servers in the group
  - Best effort multicast – delivery to only reachable servers
  - Reliable multicast – ensure the message delivered to all servers
- Failures in the middle of an atomic multicast
  - Failures of the recipient processes or the communication links:
    - The message originator uses a time-out or acknowledgements, and also decides to abort the multicast or continue by excluding the failed members from the group
  - Failure of the originator:
    - One of recipients chosen as the new originator to decide whether to abort or complete the partially completed multicast

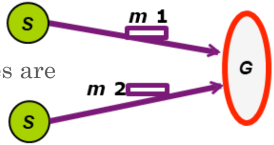


Message Delivery Ordering

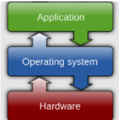
- Multiple messages multicast to the same group may arrive at different members (sites) of the group in different orders – need ordered delivery to the application processes
- Multicast orderings in increasing order of strictness:
  - FIFO, causal and total orders



- FIFO order – Multicast messages from a single source are delivered in the order they were sent
  - Assign message sequence numbers
  - Communication handler can delay messages or reject duplicates

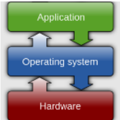


- Causal order – Causally related messages from multiple sources are delivered in their causal order
- Total order – All messages multicast to a group are delivered to all members of the group in the same order



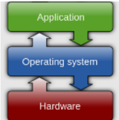
Delivery in Causal Order

- Causal ordering of messages - two messages are causally related to each other if one message is generated after the receipt of the other
  - This message order needs to be preserved at all sites
- Birman-Schiper-Stephenson Protocol - similar to vector logical clock
  - Each message is time-stamped by a sequence vector S where each entry is the number of messages received by the sender from that group member:
    $S = (S_1, S_2, \ldots, S_n)$
  - Accept a message **m** from process **i** with vector $\mathbf{T = (T_1, T_2, \ldots.., T_n)}$ if the member **j** has received all **previous messages** from **i** (that is, $\mathbf{T_i = S_i + 1}$), and the member **j** has received **all messages** also seen by **i**, (that is, $\mathbf{T_k \leq S_k}$ for all $\mathbf{k \neq i}$)
  - Delay accepting the message **m**, otherwise: if $\mathbf{T_i > S_i + 1}$ (another message from **i** is on the fly) or there exists a $\mathbf{k \neq i}$: $\mathbf{T_k > S_k}$ (this message is from the future)
  - Reject any message if $T_i \leq S_i$ (duplicate message)



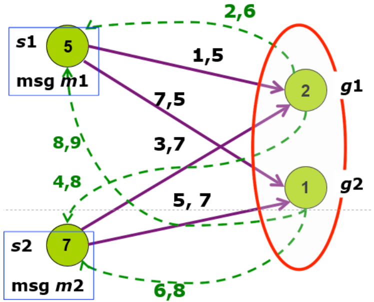
Two-Phase Total-Order Multicast

- A reliable and total order multicast is called an atomic multicast
- Two-phase total-order multicast protocol
  - Combining the atomic and total order broadcasts
  - First phase – originator broadcasts messages and collects acks with logical timestamps from all group member
  - Second phase – after all acks received, the originator sends commitment message with the highest timestamp. Receiver decides if buffer or deliver msg.
- Message originator
  - Broadcasts messages, collect acknowledgments (ack) with logical timestamps from all group members
  - Then sends a commitment message with the highest logical ack timestamp (taken as commitment timestamp)
- Recipient
  - Sends ack with the logical clock value as timestamp (local ack stamp)
  - Do not deliver a message with commit timestamp t until the commit message for all messages with local ack stamp < t has been committed – commit messages in the commitment order
  - Deliver messages in the order of the commit timestamp



Two-Phase Total-Order Multicast Example

- Two messages m_1 and m_2 broadcast between two sources (s_1, s_2) and two of the group members (g_1, g_2), with the initial logical clock times in circles

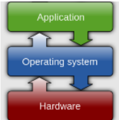


Multicast – solid lines

Acknowledgment – dashed lines

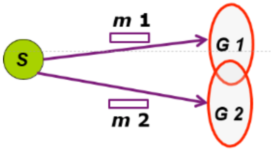
Multicast Message	Ack Time	Commit Time
m 0	2	delivered
m 1	6	9
m 2	8	8
m 3	10	pending

Buffer management in the communication handler of ***g1***

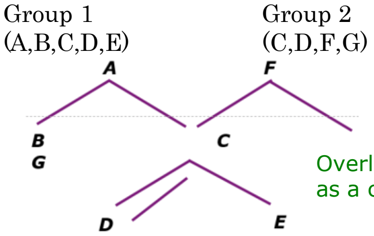


Overlapping Groups

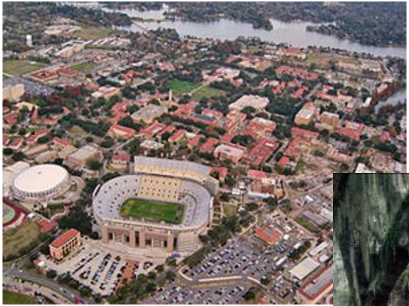
- Multicast to overlapped groups
  - A process may belong to more than one group
- Coordination among groups to maintain consistent ordering of messages:
  - Impose some agreed upon structures (a spanning tree) for the groups and multicast messages using the structures
  - A multicast message m is first sent to the group leader (root of a tree) and then to all group members by routing



Two overlapped groups



Overlap set (C,D) appears as a common subtree



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP