ID1217 Concurrent Programming
Lecture 15

ROYAL INSTITUTE
OF TECHNOLOGY

# Distributed Programming with Message Passing

Vladimir Vlassov
KTH/ICT/EECS

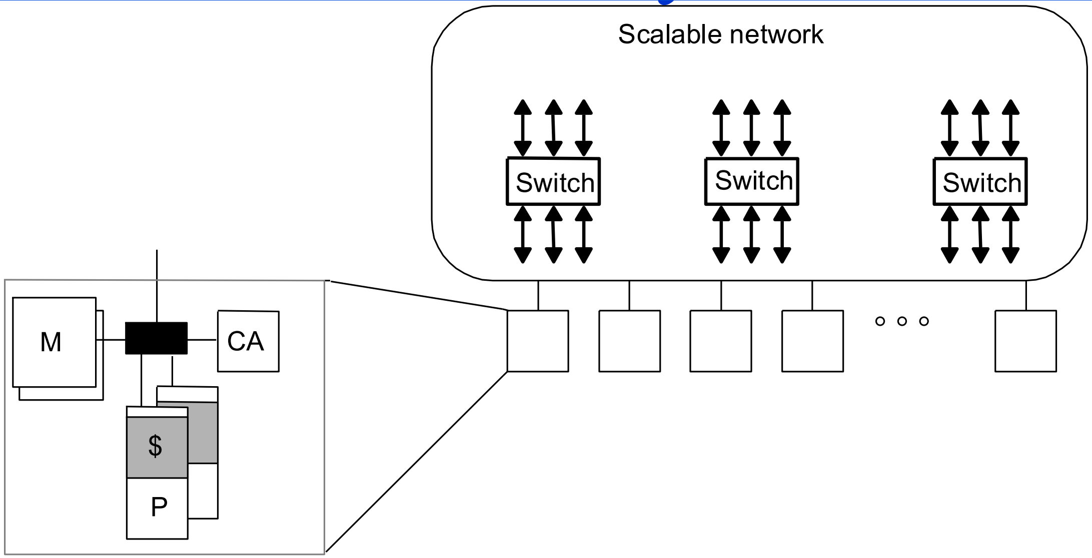
# Outline

- Introduction
- Message passing
  - Channels and messages
  - Asynchronous message passing
  - Synchronous message passing
  - Synchronous vs asynchronous message passing

# Distributed Programming

- A **distributed program** is formed of processes distributed among nodes of a distributed memory platform
  - No shared variables, only communication channels are shared
  - Processes have to exchange messages in order to interact
- **Message passing** provide ability for moving data across process memory spaces
  - Message passing primitives: **send** and **receive**
  - Distributed programs use message passing for inter-process communication
- Programming options
  - Design a special programming language or extend an existing one
  - Provide an existing (sequential) language with an external library for distributed programming, e.g. socket API, MPICH

# Distributed Memory Architecture




- A node is a single-processor or SMP (Symmetric MultiProcessor)
- No physical memory shared across nodes
- Scalable interconnection network for communication
  - Add nodes – add switches
  - Large number of independent communication paths between nodes
  - Allows a large number of concurrent network transactions
  - Bulk transfer

# Some Existing Mechanisms and Environments

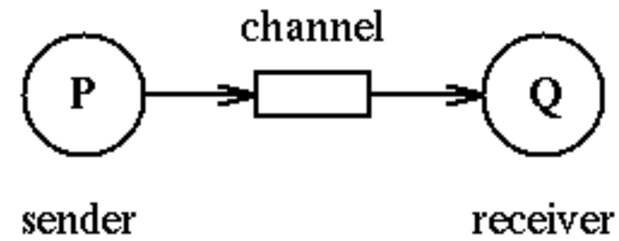
- Communication mechanisms:
  - Asynchronous message passing
  - Synchronous message passing
  - Remote procedure call (RPC) and rendezvous
  - Remote method invocation (RMI)
- Distributed programming environments and APIs:
  - Berkley Sockets API – substrate technology
  - RPC API, Java socket API, CORBA, Java RMI
  - PVM (Parallel Virtual Machine)
  - **MPI (Message Passing Interface)**
  - Pro-Active component programming environment
  - Web-service programming environments in Java EE
  - …

# Programming with Distributed Processes

- Processes of a distributed applications are heavy-weight processes started asynchronously on different (or the same) computers
  - One proc can spawn a child by `fork`
  - The child can execute a different program by `exec` (or `rexec`)
- **Single Program Multiple Data (SPMD) model**
  - One program is written for several processes (tasks)
  - Control (conditional) statements are used to select a task for each process
  - To select a task, process needs to know: the total number of processes in the application (or in a group), its own number (rank), and a problem size.
  - For example: MPICH uses SPDM
- **Multiple Program Multiple Data (MPMD) model**
  - Different programs are written for different processes (tasks)
    - For example, servers and clients
  - For example, PVM allows MPMD

# Message Passing

- **Message passing** is sending and receiving messages via communication channels shared by processes
  - Send/receive is one-way data transfer from the memory of a source (sending) proc to the memory of destination (receiving) proc




- Message passing involves synchronization - a message cannot be received until it has been sent.
- A channel can be implemented as:
  - a shared buffer – on a single processor or a shared memory multiprocessor
  - a communication link (communication HW assist) – on a distributed memory multiprocessor or network of workstations

# Message Passing Models

- **Synchronous message passing** – blocking semantics of both, send and receive
  - Send completes after matching receive is posted and data are sent
  - Receive completes after data has be received from the matching send
  - Channel can be implemented without buffering
- **Asynchronous message passing** – non-blocking send and blocking receive – send is asynchronous w.r.t. receive
  - Send completes after the send buffer may be reused
  - Channel is an unbounded FIFO queue of messages

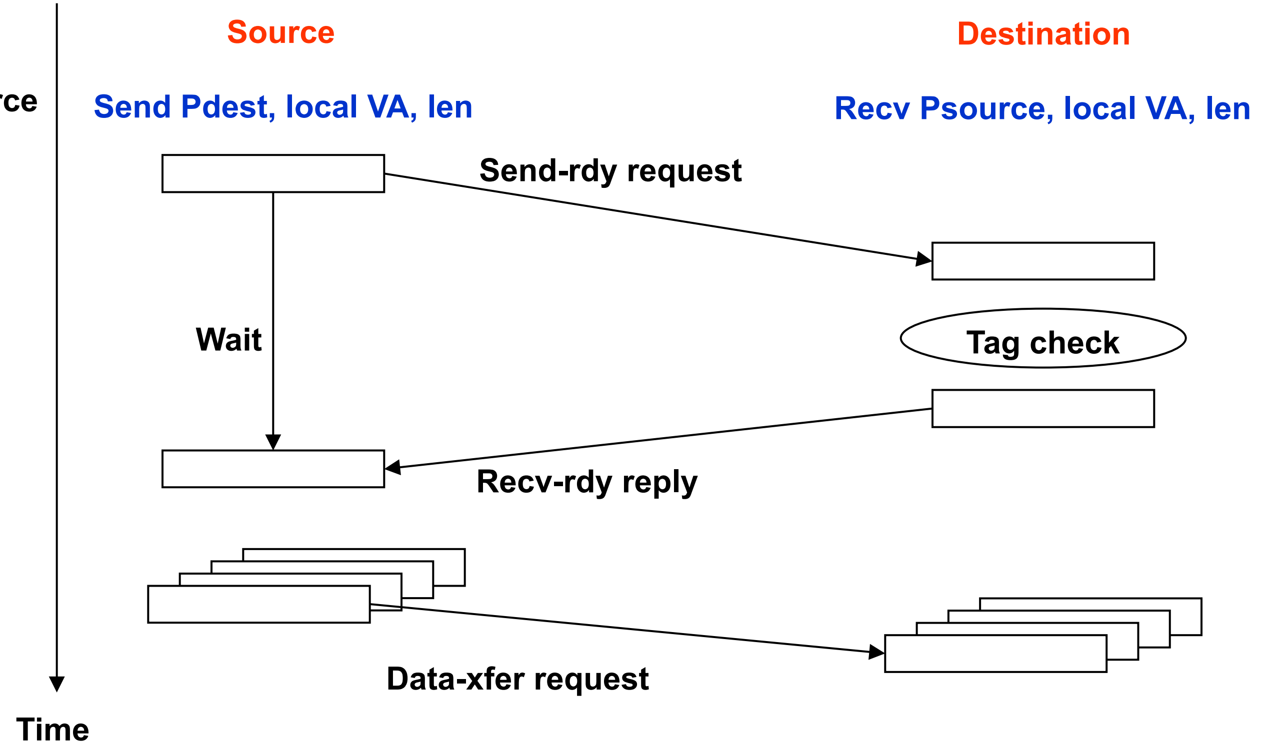
# Synchronous Message Passing

(1) Initiate send
(2) Address translation on Psource
(3) Local/remote check
(4) Send-ready request

(5) Remote check for
posted receive
(assume success)

(6) Reply transaction

(7) Bulk data transfer
Source VA → Dest VA or ID

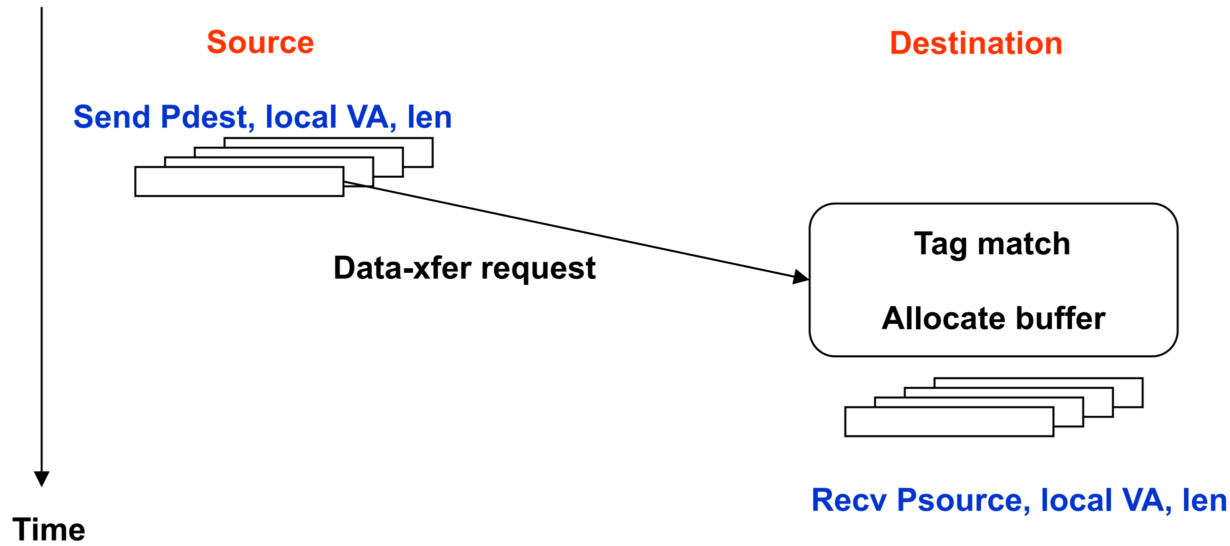



- **3-phase protocol** (match table at receiver): send-rdy, recv-rdy and data-xfer
  - 2-phase protocol (receiver-initiated, match table at sender): recv-rdy and data-xsfer. Receiver must know who will send
- Constrained programming model. Deterministic

# Asynchronous Message Passing: Optimistic

ROYAL INSTITUTE OF TECHNOLOGY

(1) Initiate send
(2) Address translation
(3) Local/remote check
(4) Send data
(5) Remote check for posted receive; on fail, allocated buffer




- **Assumes an unbounded amount of storage** outside usual process memory
  - Storage for buffering required within msg layer.
  - Problem of input buffer overflow
- **No ready-to-receive handshaking before the data transfer**
- More powerful programming model
  - Allows to avoid deadlocks
  - Wildcard receive => **non-deterministic**

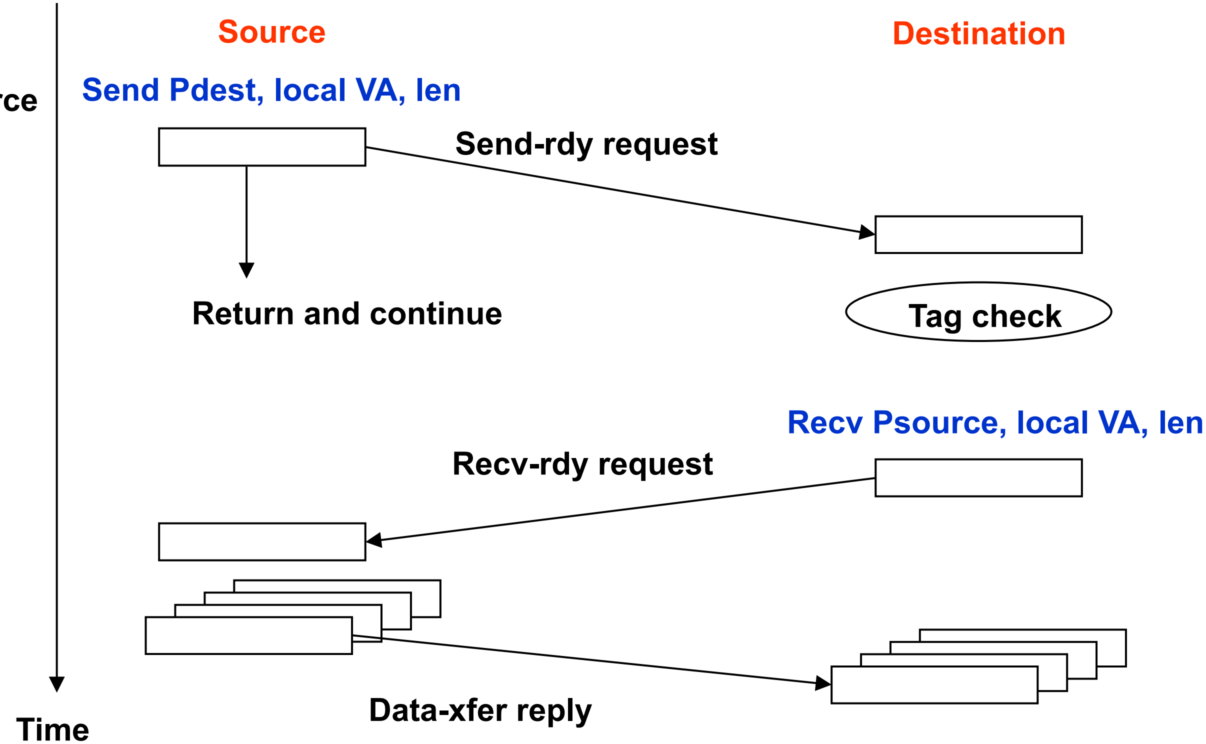
# Asynchronous Message Passing: Conservative

(1) Initiate send
(2) Address translation on Psource
(3) Local/remote check
(4) Send-ready request

(5) Remote check for
posted receive
(assume fail)
record send-ready

(6) Receive-ready request

(7) Bulk data reply
Source VA → Dest VA or ID




- **3-hase protocol with hand-shake**.
  - Send returns as soon as data are buffered out
  - Where is the buffering? First at sender then can move to receiver (if it will)
  - Short message optimizations - send without hand-shake, use destination credit (reserved space), check the credit locally before sending

# Basic Interaction Patterns in Distributed Programs

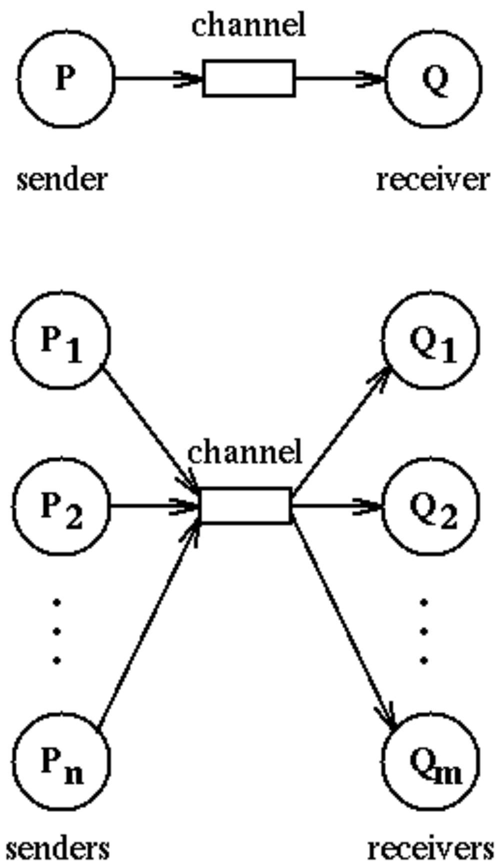
- **Producer-Consumer** (pipelines) – one way
  - Output is a function of the input and the local state
  - Receive then send
- **Client/server** – two way as master/slave
  - Client initiates and sends first, server receives first and then replies
  - Should interact with a pure request/response protocol to avoid deadlocks
  - Roles may change in particular interaction
- **Interacting peers** – two way as equals
  - Typically interaction is initiated by one of the peers
  - All should be ready to receive

# Asynchronous Message Passing

- First we consider asynchronous message passing
  - Assumes arbitrary storage "outside the local address spaces"
  - **Shared channels** – communication abstraction – unidirectional links with unbounded buffering
- Communication primitives
  - **Non-blocking send** – sends data from local memory on a specified channel
  - **Blocking receive** – delays until a message has arrived from a specified channel, receives a message and store it to a specified location

# Channels

- **Channel** is a unidirectional communication link from a sending process(es) to a receiving process(es) with an unbounded buffer for a FIFO queue of messages
  - Can be defined as bi-directional links
- Operations:
  - **Send** – append a message to a channel queue
  - **Receive** – remove a message from the head of the channel queue



# Channels (cont'd)

- Channels are known to all processes – global variables.
- A single channel:

  `chan name(type`$_1$ `field`$_1$`; ...; type`$_n$ `field`$_n$`);`

  - List of pairs `type`$_i$ `field`$_i$ defines a structure of messages transferred over the channel;
  - Optional field names can be omitted:

  `chan name(type`$_1$`; ...; type`$_n$`);`

- Array of channels:

  `chan name[n](type`$_1$`, …, type`$_m$`);`

  - Defines an array of **n** identical channels
- Examples:

```
chan input(char);
chan disk_access(int cylinder, int block, int count, char* buf);
chan result[n](int)
```

# Communication Primitives

- **Send** a message to a channel `ch`

```
send ch(expr1, …, exprn);
```

  - Non-blocking (asynchronous)
  - Evaluates expressions $expr_1, \ldots, expr_n$
  - Collects results into a message (types must match to those in the channel declaration)
  - Appends the message to the end of the channel queue `ch`
- **Receive** a message from a channel `ch`

```
receive ch(var1, …, varn);
```

  - Blocking: Delays the receiver until there is a message in `ch`
  - Extract the message at the head of the channel queue
  - Store message fields to $var_1, \ldots, var_n$
- Send and Receive are atomic
- FIFO ordering: messages are received in the same order as they were sent

# Communication Primitives (cont'd)

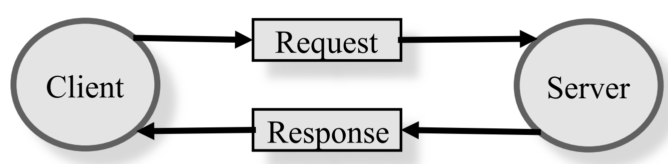
- Additional operation:

```
boolean empty(ch);
```

  - Returns true if channel ch contains no messages, otherwise returns false
  - Exposes race condition

# Clients and Servers

- **Client** process is proactive
  - Sends a request to a server process over a request channel
  - Waits until the request is serviced and a reply is sent by the server
  - Receives the reply from a reply channel
- **Server** process is reactive
  - Waits for a client request on the request channel
  - Receives and handles the request, generates a reply
  - Sends the reply to the client over the reply channel associated with a client
- In general, server's reply is a function of client's request and the server state



# Server and Clients

- The single-threaded server behaves as an active remote monitor.
- Each request type is mapped to a procedure (operation) at the server side.

```
type op_kind = enum(op1, ..., opn);
type arg_type = union(arg1, ..., argn);
type result_type = union(res1, ..., resn);
chan request(int clientID, op_kind, arg_type);
chan reply[n](res_type);
process Server {
  int clientID; op_kind kind; arg_type args;
  res_type results; declarations of other variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, kind, args);
    if (kind == op1)
      { body of op1; }
    ...
    else if (kind == opn)
      { body of opn; }
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
  arg_type myargs; result_type myresults;
  place value arguments in myargs;
  send request(i, opj, myargs);    # "call" opj
  receive reply[i](myresults);     # wait for reply
}
```

# Order of Servicing Client Requests

- Order of servicing may depend on the state of the server and on a client request (e.g. type of request, request parameters)
- **FCFS** – first come first served – a simple server architecture without scheduling.
- **“Best” request is serviced first (next)**
  - Best: shortest, largest, system, etc.
  - To choose the current best, server must receive and examine all requests pending in a channel
  - The server maintains a list of received requests – a request list
  - Requests are scheduled from the list in desired order
  - Two threads can do this:
    - One (receiving thread) receives and places requests on the request list
    - Another (servicing/replying thread) fetches a request from the list and services it

# Example: A File Server

- Illustrates a multithreaded server and **session** communication – **conversational continuity**
- File server is a part of DFS (distributed file system)
  - Provides for remote clients access to external files on the server's disk
- Client requests:
  - OPEN access to a file – also opens a session
  - READ/WRITE the file
  - CLOSE the file – also closes the session
- Multi-processed (multithreaded) server – to run several concurrent sessions
- A server keeps a session with a client open until it receives CLOSE request from the client

# File Server Processes and Channels

- `n` server processes – a scalable session-oriented file server

  ```
  process File_Server[n]
  ```

  - Allow servicing `n` clients (access `n` files) simultaneously
- Request channels
  - One channel for OPEN requests

    ```
    chan open(fname, clientID)
    ```

  - `n` channels for access (READ/WRITE/CLOSE) requests

    ```
    chan access[n](kind, args)
    ```

- Replies channels
  - `m` channels for OPEN replies (one per client)

    ```
    chan open_reply[m](serverID)
    ```

  - `m` channels for access replies (one per client)

    ```
    chan access_reply[m](results)
    ```

ROYAL INSTITUTE OF TECHNOLOGY

# Client – Server Interaction

- Rendezvous part –opens file and opens session
  - Client with `myid`:

```
send open(fname, myid);
receive open_reply[myid](serverID);
```

  - Server with `myid` which services a client with `clientID`

```
receive open(fname, clientID);
open the file;
if (successful)
    send open_reply[clientID](myid);
```

- Conversation part – **session** – request-response interaction
  - one client to one server, using `clientID` and `serverID`

ROYAL INSTITUTE OF TECHNOLOGY

# File Server and Clients

- Channels and file server processes

```
type kind = enum(READ, WRITE, CLOSE);
chan open(string fname; int clientID);
chan access[n](int kind, types of other arguments);
chan open_reply[m](int serverID);  # server id or error
chan access_reply[m](types of results);  # data, error, ...

process File_Server[i = 0 to n-1] {
  string fname; int clientID;
  kind k; variables for other arguments;
  bool more = false;
  variables for local buffer, cache, etc.;
  while (true) {
    receive open(fname, clientID);
    open file fname; if successful then:
    send open_reply[clientID](i); more = true;
    while (more) {
      receive access[i](k, other arguments);
      if (k == READ)
        process read request;
      else if (k == WRITE)
        process write request;
      else   # k == CLOSE
        { close the file; more = false; }
      send access_reply[clientID](results);
    }
  }
}
```

# File Server and Clients (cont'd)

- Client processes:

```
process Client[j = 0 to m-1] {
  int serverID;   declarations of other variables;
  send open("foo", j);     # open file "foo"
  receive open_reply[j](serverID);  # get back server id
  # use file then close it by executing the following
  send access[serverID](access arguments);
  receive access_reply[j](results);
   ...
}
```

# Use of Global Channels

- In the File Server example, **n** server processes use one global channel **open** for OPEN requests
- What if cannot have a global channel with multiple receivers?
  - usually each channel has exactly one receiver
- Use a separate manager process that allocates a server process to a client

```
receive open(...);
```

  Pick a free server process (or create a new one);
  forward the open request to it

  - Need "done" messages from servers to keep track of free servers
  - This implies that the manager receives two kinds of messages: (1) opens from clients and (2) dones from servers

# Interacting Peers

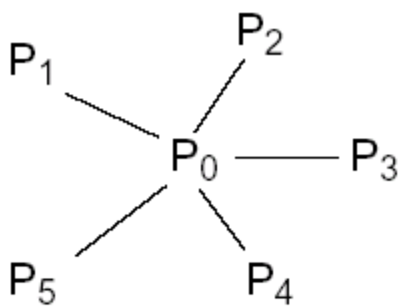
- Two way as equals: peer-to-peer
  - Server-to-server
  - Business-to-business
  - Client-to-client
- Environments and applications:
  - (Un)Structured overlay networks, DHT (distributed hash tables)
  - File sharing,
  - P2P distributed services (e.g. naming service, etc.)
  - Scalable P2P-based services in distributed systems
  - Distributed games

# Example: Exchange of Values

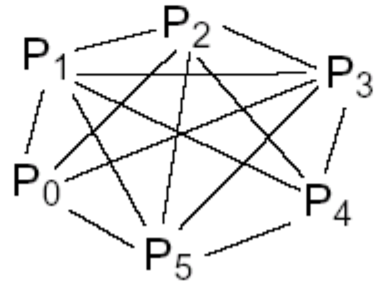
- A typical problem in peer-to-peer computing.
- Assume, ***n*** processes; each has a value; want every process to learn every value
- Assume a collective reduction operation: every process needs to know the smallest and the largest of the values distributed among processes
- Trade-off: Number of messages

# Solution Structures

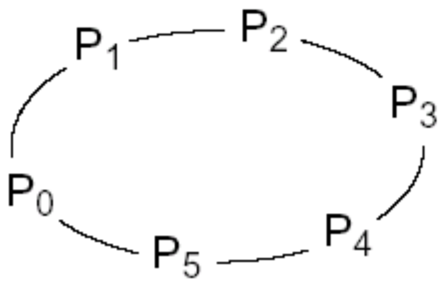
a) Centralized – star configuration, with coordinator in center
b) Symmetric – fully-connected (complete) graph
c) Ring – closed (circular) pipeline




(a) Centralized solution




(b) Symmetric solution




(c) Ring solution

# (a) Centralized Solution

- One process collects all values, does all the "work" and broadcasts result to others
- In total **2(*n*-1)** messages over ***n*+1** channels
  - With a "broadcast" message – *n* distinct messages

```
chan values(int), results[n](int smallest, int largest);
process P[0] {   # coordinator process
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # gather values and save the smallest and largest
  for [i = 1 to n-1] {
    receive values(new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
  # send the results to the other processes
  for [i = 1 to n-1]
    send results[i](smallest, largest)
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  send values(v);
  receive results[i](smallest, largest);
}
```

# (b) Symmetric Solution

- Each process sends its value to others, collects all values from others and does the "work" individually
  - In total ***n*(*n*-1)** messages over ***n*** channels
  - With a "broadcast" message – ***n*** distinct messages

```
chan values[n](int);
process P[i = 0 to n-1] {
  int v;   # assume v has been initialized
  int new, smallest = v, largest = v;  # initial state
  # send my value to the other processes
  for [j = 0 to n-1 st j != i]
    send values[j](v);
  # gather values and save the smallest and largest
  for [j = 1 to n-1] {
    receive values[i](new);
    if (new < smallest)
      smallest = new;
    if (new > largest)
      largest = new;
  }
}
```

# c) Circular Pipeline

- Two consecutive rounds:

  (1) Compute current min and max in a pipeline fashion;

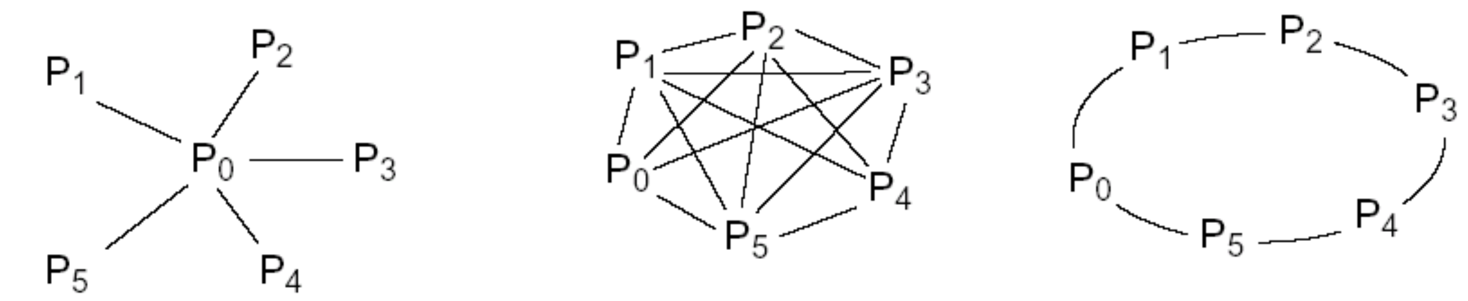
  (2) Distribute results along the ring
- In the first round, each process P[i]
  - Receives two values (current min and max) from its predecessor P[i-1]
  - Compare the received values with its own value, and
  - Sends result of comparison, min and max, to its successor P[(i+1) mod n]
  - To avoid deadlock in the first round – P[0] first sends and then receives
- In the second round, each process P[i]
  - Receives two values (final min and max) from its predecessor P[i-1]
  - Stores the values locally and sends them to its successor P[(i+1) mod n]

# c) Circular Pipeline (cont'd)

- Employs **2*n*-1** messages over ***n*** channels
- No broadcast

```
chan values[n](int smallest, int largest);
process P[0] {  # initiates the exchanges
  int v;   # assume v has been initialized
  int smallest = v, largest = v;  # initial state
  # send v to next process, P[1]
  send values[1](smallest, largest);
  # get global smallest and largest from P[n-1] and
  #   pass them on to P[1]
  receive values[0](smallest, largest);
  send values[1](smallest, largest);
}
process P[i = 1 to n-1] {
  int v;   # assume v has been initialized
  int smallest, largest;
  # receive smallest and largest so far, then update
  #   them by comparing their values to v
  receive values[i](smallest, largest)
  if (v < smallest)
      smallest = v;
  if (v > largest)
      largest = v;
  # send the result to the next processes, then wait
  # to get the global result
  send values[(i+1) mod n](smallest, largest);
  receive values[i](smallest, largest);
  if (i < n-1)
     send values[i+1](smallest, largest);
}
```

# Summary of Peer-to-Peer Communication Patterns




(a) Centralized solution (b) Symmetric solution (c) Ring solution

- All-to-all communication – **O($n^2$)** messages, but a symmetric solution is easy to program when all know all
- Star and ring configurations – **O($n$)** messages
- The circular pipeline (ring) configuration – can be <u>inefficient</u>
  - In the example, messages circulate around the pipeline two full circles before every process has learned the global result

# Synchronous Message Passing

- Blocking semantics of both, send and receive
  - Send completes after a matching receive has been posted and the source has sent data
  - Receive completes after data transfer from the matching send completes
  - Synchronization via a handshake before bulk data transfer – 3 phase protocol: send_rdy, receive-rdy, transfer.
- Synchronous send over a channel:

```
synch_send ch(expr_1, …, expr_n);
```

- Channels can be implemented without buffering
- Synchronous communication was introduced and formalized by Tony Hoare in 1978 via CSP (Communicating Sequential Processes).

# Synchronous Message Passing Limits Concurrency

- Assume, Producer and Consumer communicate with synchronous message passing:
  – Producer and Consumer arrive at communication stage at different times;
  – This causes each pair of send/receive to delay, and, as consequence, the total execution time to increase

```
channel values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    do some computation;
    synch_send values(data[i]);
  }
}

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    do some computation;
  }
}
```

- Another example: Client-Server with synch. message passing:
  – Unnecessary delays:
    - When Client wants to releases resource managed by Server
    - When Client issues a write request to an output device controlled by Server that does not need to be acknowledged

# Synchronous Message Passing is Deadlock Prone

- Example: two processes exchange values
- A symmetric solution with synchronous send leads to a deadlock
  - The deadlock would not occur with asynchronous sends

```
channel in1(int), in2(int);
process P1 {
  int value1 = 1, value2;
  synch_send in2(value1);
  receive in1(value2);
}
process P2 {
  int value1, value2 = 2;
  synch_send in1(value2);
  receive in2(value1);
}
```

# Asynchronous vs Synchronous Message Passing

- **Asynchronous message passing:**
  - Pros
    - More convenient for programming;
    - More concurrency; less deadlock situations
  - Cons:
    - Assumes an unbounded amount of storage outside process memory
    - More complicated to implement: the amount of memory space for channels is unpredictable (and so must be allocated dynamically)
- **Synchronous message passing:**
  - Pros
    - Channels can be implemented without intermediate buffering
  - Cons:
    - Limits concurrency; deadlock prone