Distributed Programming

- Processes don't share memory.
- Processes share *channels* over which messages are passed
  - ∗ Channels may be
    - · Global,
    - · receiver specific,
    - · or sender & receiver specific.
  - ∗ One or two way.

Distributed Paradigms

Filter Data translator— Read input stream, write to output stream.

Client Active (triggering) process— Request service, often wait for response.

Server Reactive process— Wait for request, respond.

Peer Cooperating process.jjjjjjj

Asynchronous Message Passing

---

chan c(type1 v1, type2 v2 ...);

send c(x1, x2 ...);

receive c(y1, y2 ...);

empty(c)

- Channels are considered unbounded FIFO queues
- Non-blocking **send**.
- Blocking **receive**.
- **send** and **receive** generalize V and P by adding data.

Since **receive** is the only blocking call, deadlock can only occur there.

Translation:

$$\begin{array}{lll} \textbf{send}\ c(d) & \Longrightarrow & \langle c := \mathrm{concatenate}(c, [d]) \rangle \\ \textbf{receive}\ c(d) & \Longrightarrow & \langle \textbf{await}(\neg\, \mathrm{empty}(c))\ d, c := \mathrm{head}(c), \mathrm{tail}(c) \rangle \end{array}$$

Filter: Mergesort

Problem: Sort a list of values

Solution: Network filters in a tree structure

```
process Merge {
    int v1, v2;
    receive in1(v1);
    receive in2(v2);

    while ( !(v1 == EOS and v2 == EOS) ) {
        if (v2 == EOS or v1 != EOS and v1 <= v2) {
            send out(v1); receive in1(v1); }
        else{## v1 == EOS or v2 != EOS and v2 < v1
            send out(v2); receive in2(v2); } }
    send out(EOS); }
```

Client-Server.

Simulating a monitor using AMP.

- Clients send operation name and parameters and receive result
- Server receives requests and sends results
- Monitor invariant becomes loop invariant of the server.

---

process Server() {
    declare data
    initialize data
    while(true) { ## M
        receive request(clientID, opKind, parameters)
        case(opKind) {
        when OP$_0$ { *op*$_0$ *body. Calculate result*
            send reply[clientID](result) ; }
        ...
        when OP$_k$ { *op*$_k$ *body. Calculate result*
            send reply[clientID](result) ; } } }

Conditions — Signal and Urgent (LIFO) Wait

Each condition c becomes a queue q_c local to the server.

```
procedure op_i() {
        W
        wait c ;
        X
    return result; }
```

```
procedure op_j() {
        Y
        signal c ;
        Z
        return result; }
```

Becomes

```
case( opKind ) { ...
when OP_i {
  W
  q_c.insert( clientID ) ; }
```

```
when OP_j {
  Y
  if( ! q_c.empty() ) {
    q_c.remove( id ) ;
    X'
    send reply[id]( result' ) ; }
  Z
  send reply[clientID]( result ) ; }
... }
```

Considerations

- Local data may have to be put on queue

- Multiple wait(c) in monitor: queue indication of which wait.
- Multiple signal(c) in monitor: write subroutine for signal.

Session-based: client-server

Each client has the undivided attention of a server for as long as it needs.

Example:

```
process Server [ i = 1 to N ] {
    while( true ) {
        int clientID ;
        receive openChan( clientID )
        send replyChan[ clientID ]( openAck( i ) )
        State serverState := initState ;
        do {
            Request request ;
            receive requestChan[i]( request ) ;
            Reply reply ;
            ... compute reply and change state...
            send replyChan[ clientID ]( reply ) ;
        } while( ...session not over... ) ; }
    }
```

Interacting Peers: Exchanging Values

Task: Determine the largest and smallest value held by processes.

Centralized: Coordinator gathers all, and sends results.

- Asymmetric — coordinator does all the work
- $2(\mathtt{n}-1)$ messages, $\mathtt{n}$ channels

Symmetric: Each sends data to all others, receives from all others, then computes results.

- $\mathtt{n}(\mathtt{n}-1)$ messages, $2\mathtt{n}$ channels

Logical Ring: Recv local max, min from prev; Send local max, min to next; Recv global max, min from prev; Send global max, min to next.

- $2(\mathtt{n}-1)$ messages, $\mathtt{n}$ channels

AMP in Java – Sockets

- Two-way channels for bytes.
- ServerSocket – allocates a port for the channel.
- Socket – opens a channel on the port.
  - ∗ inputStream
  - ∗ outputStream

(Disclaimer: Don't take the following code too literally. I've omitted some necessary exception handling.)

Multi-session Server

```
void startSession(final Socket socket) {
    Thread sessionThread = new Thread() {
        public void run() {
            BufferedReader from_client
                = new BufferedReader(
                    new InputStreamReader(
                        socket.getInputStream()));
            PrintWriter to_client = new PrintWriter(
                socket.getOutputStream());
            ... communicate with client ....
            to_client.close();
            from_client.close();
            socket.close(); } }
    sessionThread.start() ; }
...
ServerSocket listen = new ServerSocket(8080);
while( true ) {
    Socket socket = listen.accept();
    startSession( socket ) ; }
```

Client

```
InetAddress serverHost
        = InetAddress.getByName( "web.engr.mun.ca" ) ;
Socket socket = new Socket(serverHost, 8080);
BufferedReader from_server = new BufferedReader(
        new InputStreamReader(
              socket.getInputStream()));
PrintWriter to_server = new PrintWriter(
        socket.getOutputStream());
... use socket to communicate with server...
from_server.close();
to_server.close();
socket.close();
```

Synchronous Message Passing

- Non-buffered communication
- `sync_send` blocks until message is received
- Combined communication and synchronization
- Can be viewed as distributed assignment statement.
  - Often reduces concurrency — sender or receiver waiting.
- More prone to deadlock.

Examples

- Pipelined sieve of Eratosthenes
  - First number received, $\mathtt{p}_i$, is prime
  - From remaining values, pass on only if $\mathtt{x}\%\mathtt{p}_i \neq 0$
- Heartbeat compare and exchange sort
  - $*$ Each of k processes holds n/k data elements
  - $*$ Even rounds:
    - · if $\mathtt{i}$ is even, $\mathtt{P}[\mathtt{i}]$ sends its largest to $\mathtt{P}[\mathtt{i}+1]$, receives from $\mathtt{P}[\mathtt{i}+1]$ its smallest.
    - · if $\mathtt{i}$ is odd, $\mathtt{P}[\mathtt{i}]$ sends its smallest to $\mathtt{P}[\mathtt{i}-1]$, receives from $\mathtt{P}[\mathtt{i}-1]$ its smallest.
  - $*$ Odd rounds:
    - · if $\mathtt{i}$ is odd, $\mathtt{P}[\mathtt{i}]$ sends its largest to $\mathtt{P}[\mathtt{i}+1]$, receive from $\mathtt{P}[\mathtt{i}+1]$ its smallest.
    - · if $\mathtt{i}$ is even, $\mathtt{P}[\mathtt{i}]$ sends its smallest to $\mathtt{P}[\mathtt{i}-1]$, receive from $\mathtt{P}[\mathtt{i}-1]$ its largest.

```
process HeartBeatSort[ i : 0 to k-1] {
     ...find largest and smallest of local items...
    int round := 0 ;
     while( ... )
         if( (i+round) is even and i < k-1 ) {
             send c[i+1]( largest ) ;
             receive c[i]( largest ) ; }
         else if(  (i+round) is odd and i > 0 ) {
             send c[i-1]( smallest ) ;
             receive c[i]( smallest ) ; }
         ...recalculate largest and smallest ...
         round += 1 ; } }
```

Termination is a bit tricky. We need to run at least $2(n/k+k-1)$ rounds, but then have to make sure that the last exchange that each pair makes is not counterproductive.