

# Programming Paradigms

## Lecture 10

---

**Slides are from Prof. Chin Wei-Ngan from NUS**

More on Declarative Concurrency

# Agents and Message Passing Concurrency

# Client-Server Architectures

- Server provides some service
  - receives message
  - replies to message
  - examples: web server, mail server, ...
- Clients know address of server and use service by sending messages
- Server and clients run independently

# Client-Server Applications ...

- With declarative programming, it is impossible to write a client/server program where the server does not know which client will send the next message.
- **Observable *nondeterministic* behavior:** the server can receive information in any order from two independent clients.
- The server has **only** an input stream from which it reads commands.

# The Message-Passing Concurrent Model

- Extends the declarative concurrent model by adding one new concept, an **asynchronous communication channel**.
- Any client can send messages to the channel at any time and the server can read all the messages from the channel (no limitations).
- A client/server program may give different results on different executions because the order of clients' sends is not fixed.
- **Message-passing model** is *nondeterministic* and therefore no longer declarative.

# Peer-to-Peer Architectures

- Similar to Client-Server:
  - every client is also a server
  - communicate by sending messages to each other
- We call all these guys (client, server, peer)  
*agent*
- In [van Roy, Haridi; 2004] book, this is called  
*portObject*

# Common Features

## ■ Agents

- have identity                      mail address
- receive messages                mailbox
- process messages                ordered mailbox
- reply to messages                pre-addressed return letter

## ■ Now how to cast into programming language?

# Message Sending

- Message              data structure
- Address              **port**
- Mailbox              stream of messages
- Reply                dataflow variable in message

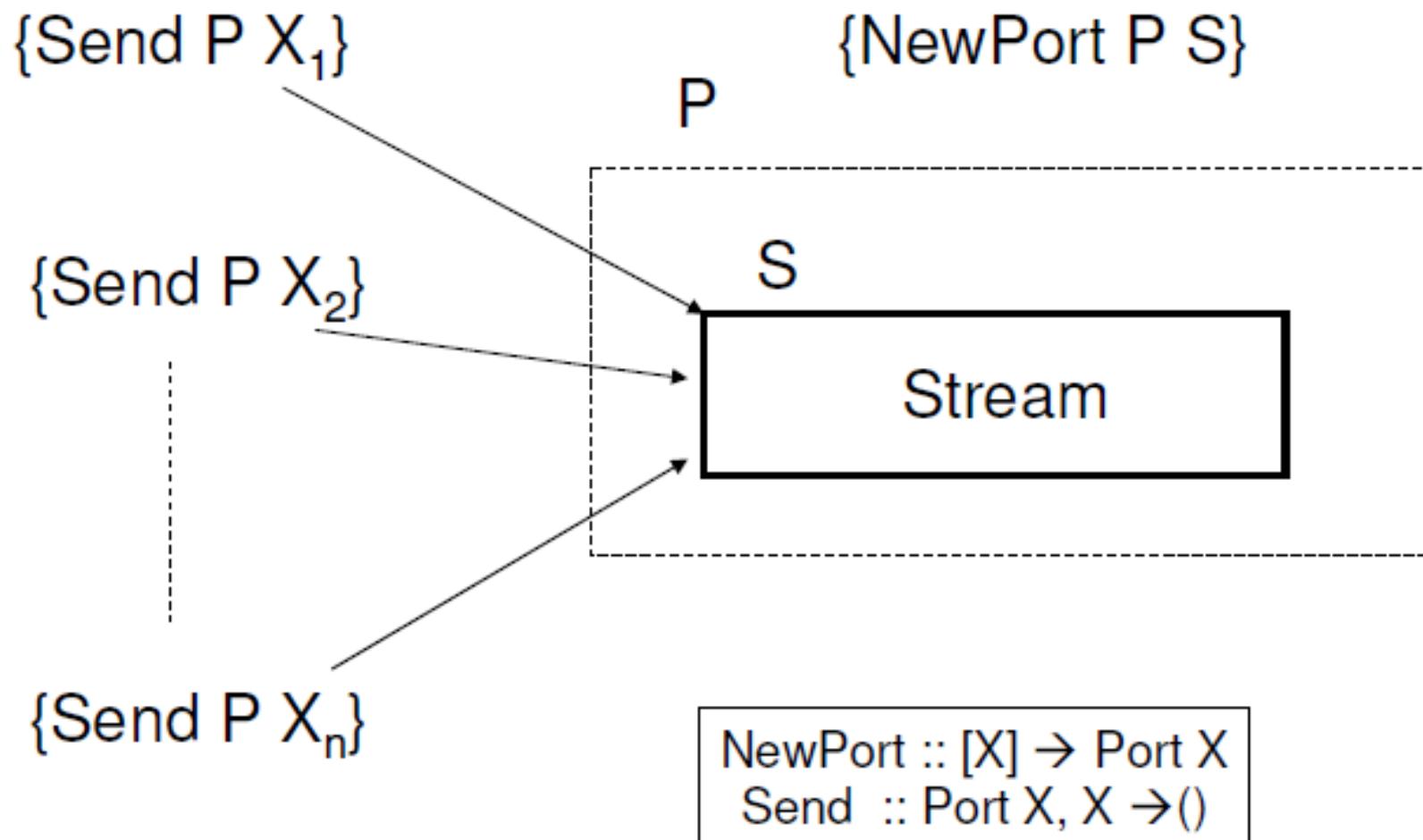
Type :: Port X

message type

# Ports

- A **port** is an ADT with two operations:
  - $\{ \text{NewPort } S \ P \}$  or equivalently  $P = \{ \text{NewPort } S \}$ : create a new port with entry point (channel)  $P$  and stream  $S$ .
  - $\{ \text{Send } P \ x \}$ : append  $x$  to the stream corresponding to the entry point  $P$ .
- Successive sends from the same thread appear on the stream in the same order in which they were executed.
- This property implies that a port is an asynchronous FIFO (first-in, first-out) communication channel.

# Port and its Stream



# Ports

- **Asynchronous:** a thread can send a message at any time and it does not need to wait for any reply.
- As soon as the message is in the communication channel, the thread can continue executing.
- Communication channel can contain many pending messages, which are waiting to be handled.

# Example

```
declare S P  
P={NewPort S}  
{Browse S}
```

- Displays initially `s<future>` (or `_`)

# Example

```
declare S P  
P={NewPort S}  
{Browse S}
```

- Execute {Send P a}
- Shows a |\_<future>

# Example

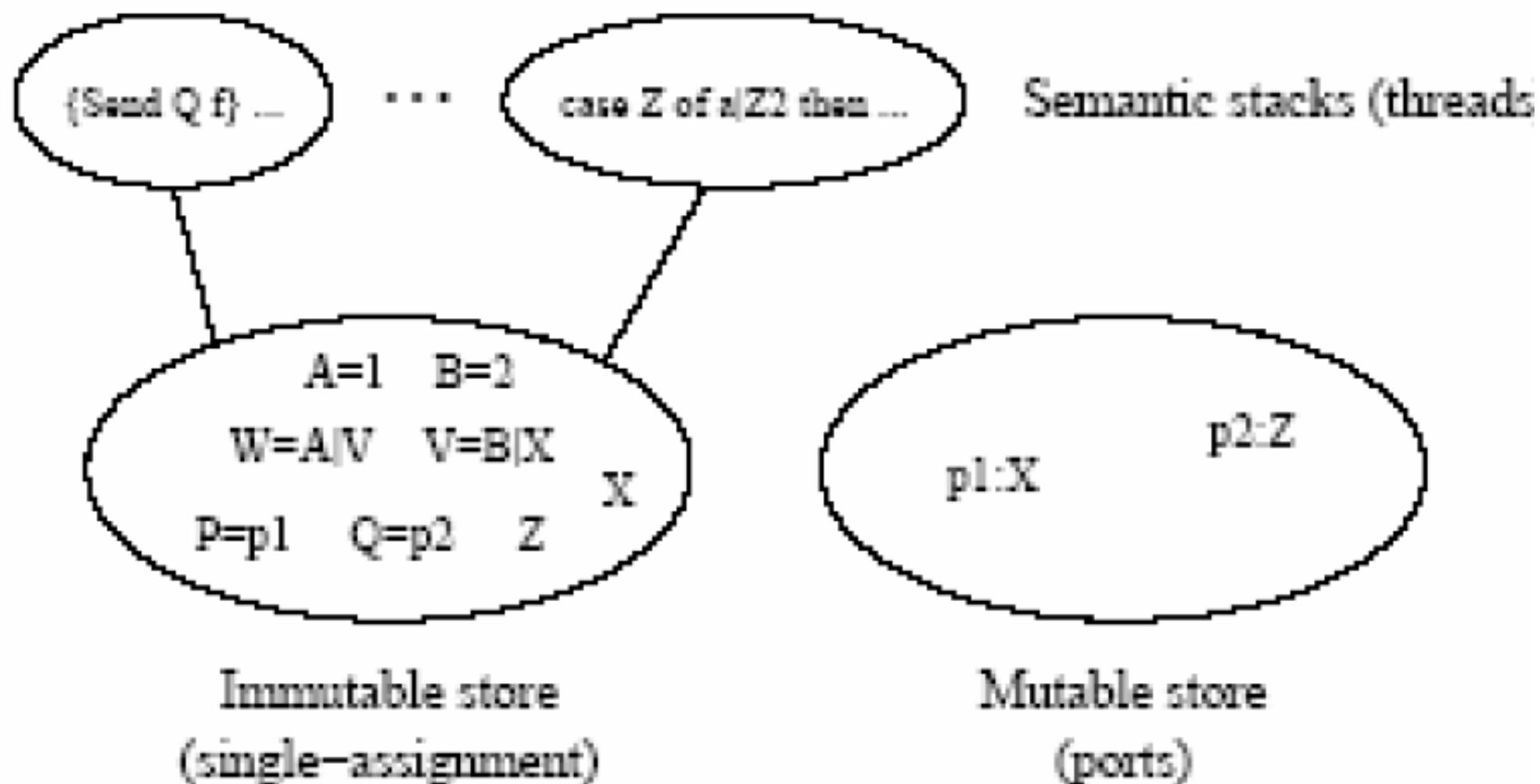
```
declare S P  
P={NewPort S}  
{Browse S}
```

- Execute {Send P b}
- Shows a|b|\_<future>
- Note that {Send P a} and {Send P b} are in the same thread

# Semantics of Ports

- Extend the execution state of the declarative model by adding a **mutable store**  $\mu$
- This store contains ports, i.e. pairs of the form  $x : y$ , where  $x$  and  $y$  are variables of the single-assignment store ( $x$  is the channel's name and  $y$  is the *current last position of stream*).
- The mutable store is initially empty.
- The semantics guarantees that  $x$  is *always bound* to a name value that represents a port and that  $y$  is *unbound*.
- The execution state becomes a triple  $(MST, \sigma, \mu)$  (or  $(MST, \sigma, \mu, \tau)$  if the trigger store is considered).

# The Message-Passing Concurrent Mode



# The NewPort Operation

- The semantics of  $(\{\text{NewPort } \langle x \rangle \langle y \rangle\}, E)$  is:
  - Create a **fresh port name** (also called **unique address**)  $n$ .
  - Bind  $E(\langle y \rangle)$  and  $n$  in the store.
  - If the binding is successful, then add the pair  $E(\langle y \rangle) : E(\langle x \rangle)$  to the mutable store  $\mu$ .
  - If the binding fails, then raise an error condition.

# The Send Operation

- The semantics of  $(\{\text{Send } \langle x \rangle \langle y \rangle\}, E)$  is:
  - If the activation condition is true ( $E(\langle x \rangle)$  is determined), then:
    - If  $E(\langle x \rangle)$  is not bound to the name of a port, then raise an error condition.
    - If the mutable store contains  $E(\langle x \rangle) : z$ , then:
      - Create a new variable  $z0$  in the store.
      - Update the mutable store to be  $E(\langle x \rangle) : z0$ .
      - Create a new list pair  $E(\langle y \rangle) | z0$  and bind  $z$  with it in the store.
    - If the activation condition is false, then suspend execution.

# Question

```
declare S P  
P={NewPort S}  
{Browse S}  
thread {Send P a} end  
thread {Send P b} end
```

- What will the Browser show?
- Note that each {Send P ...} is in a separate thread

# Question

```
declare S P  
P= {NewPort S}  
{Browse S}  
thread {Send P a} end  
thread {Send P b} end
```

- Which will the Browser show?
- Either
  - a | b | \_<future> or
  - b | a | \_<future>
- non-determinism: we can't say what

# Answering Messages

- Traditional view
- Include the entry port  $P'$  of the sender in the message:  
 $\{\text{Send } P \text{ pair (Message } P')\}$
- Receiver sends answer message to  $P'$   
 $\{\text{Send } P' \text{ AnsMessage}\}$

# Answering Messages

- Do not reply by address, use something like pre-addressed reply envelope
  - dataflow variable!!!
- {Send P pair(Message Answer) }
- Receiver can bind Answer!

# Port Objects

- A **port object** is a combination of one or more ports and a stream object.
- This *extends stream objects* in two ways:
  - First, many-to-one communication is possible: many threads can reference a given port object and send to it independently.
    - This is not possible with a stream object because it has to know where its next message will come from.
  - Second, port objects can be embedded inside data structures (including messages).
    - This is not possible with a stream object because it is referenced by a stream that can be extended by just one thread.

# Port Objects. Distributed Algorithm

```
declare P1 P2 ... Pn in
local S1 S2 ... Sn in
  {NewPort S1 P1}
  {NewPort S2 P2}
  ...
  {NewPort Sn Pn}
thread {RP S1 S2 ... Sn} end
end
```

- The thread contains a recursive procedure RP that reads the port streams and performs some action for each message received.
- Sending a message to the port object is just sending a message to one of the ports.
- Similar terms: **agent**, **process** (Erlang), **active object**

# A Math Agent

```
proc {Math E}
    case E
        of add(N M Answer) then Answer=N+M
        [] mul(N M Answer) then Answer=N*M
        [] int(Formula Answer) then
            Answer = ...
    end
end
```

- **Remark:** Answer is included in the stream's element  
x of {Send EntryPoint X}

# Making the Agent Work (Port Creation)

```
local S in
    MP = {NewPort S}
    proc {MathProcess Ms}
        case Ms of M|Mr then
            {Math M}
            {MathProcess Mr}
        end
    end
    thread {MathProcess S} end
end
```

- `MathProcess` is a recursive procedure that reads the port streams and performs some action for each message received.

# Making the Agent Work (Sending a Message)

```
declare A B  
thread % client 1  
  {Send MP add(2 3 A)}  
  {Browse A}  
end  
thread % client 2  
  {Send MP mul(2 3 B)}  
  {Browse B}  
end
```

- A and B are two dataflow variables which will be bound in port MP

# Recall Higher-Order Construct

```
ForAll :: {[X], X→()} → ()
```

```
proc {ForAll Xs P}  
  case Xs  
    of nil  then skip  
    [] X|Xr then {P X} {ForAll Xr P}  
  end  
end
```

- Call procedure  $P$  for all elements in  $Xs$

# Smells of Higher-Order...

- Using `ForAll`, we have

```
proc {MathProcess Ms}
    {ForAll Ms Math}
end
```

# Making the Agent Work

```
declare MP in
local S in
  MP = {NewPort S}
  thread {ForAll S Math} end
end
```

# Making the Agent Work

```
declare MP in
local S in
  MP = {NewPort S}
  thread for M in S do {Math M} end end
end
```

- The stream `s` is private (`local`) to the port.
- `Math` is associated to the port `MP`
- `MP` and `Math` can become arguments of a generic function.

# Smells Even Stronger...

- Programming with port objects can be abstracted into a function

```
fun {NewAgent Process}
```

```
    Port Stream
```

```
NewAgent :: {X→()} → Port X
```

```
in
```

```
    Port = {NewPort Stream}
```

```
    thread {ForAll Stream Process} end
```

```
    Port
```

```
end
```

- So, the previous port creation is equivalent with:

```
MP = {NewAgent Math}
```

# Why Do Agents/ Processes Matter?

- Model to capture communicating entities
- Each agent is simply defined in terms of how it replies to messages
- Each agent has a thread of its own
  - no screw-up with concurrency
  - we can easily extend the model so that each agent has a state (encapsulated)
- ***Extremely useful to model distributed systems!***

# Summary so far

- Ports for message sending
  - use stream (list of messages) as mailbox
  - port serves as unique address
- Use agent abstraction
  - combines port with thread running agent
  - simple concurrency scheme
- Introduces non-determinism... and state!

# Protocols

# Protocols

- **Protocol:** is a set of rules for sending and receiving messages
  - programming with agents
- Most well-known protocols:
  - the Internet protocols (TCP/IP, HTTP, FTP, etc.)
  - LAN (Local Area Network) protocols such as Ethernet and DHCP (Dynamic Host Connection Protocol), ...

# RMI (Remote Method Invocation)

- It seems to be the most popular of the simple protocols.
- It allows an object to call another object in a different operating system process, either on the same machine or on another machine connected by a network.
- **RMI** is a descendant of the **RPC** (Remote Procedure Call), which was invented in 1980, before object-oriented programming became popular.
- RMI became popular once objects started replacing procedures as the remote entities to be called.
- We assume that a “*method*” is simply what a port object does when it receives a particular message.

# Differences between RPC and RMI

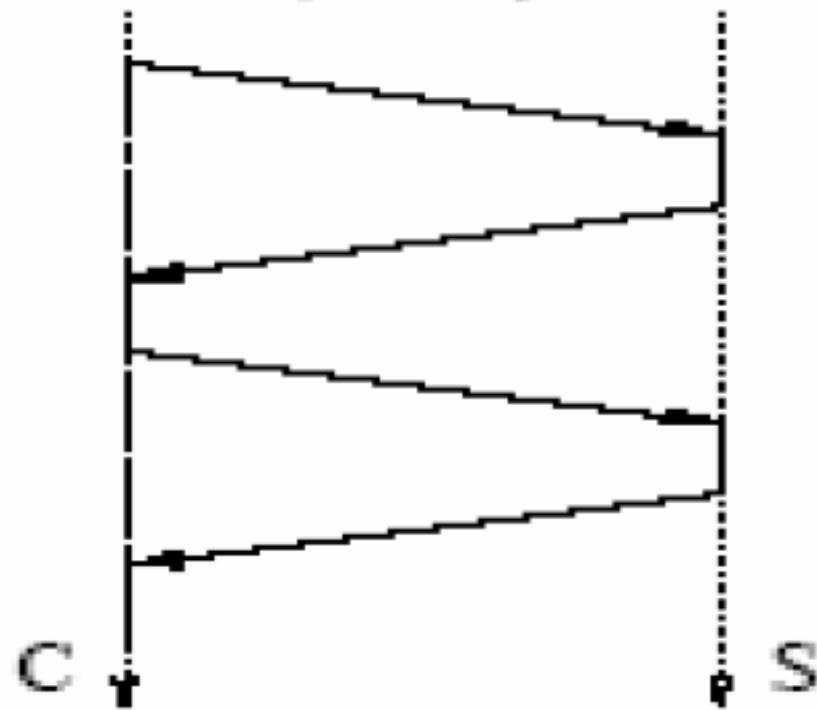
## RPC

- Faster than RMI
- Depends on the platform
- Has to convert the arguments between architectures so that each computer can use its native datatype

## RMI

- Is part of Java's object-oriented approach
- Allows multiple-concurrent method invocation
- Is portable (doesn't depend on the operating system)
- Good security system
- To call outside methods, RMI needs JNI, JDBC, RMI-IIOP, RMI-IDL, etc.

## 1. RMI (2 calls)



- A client sends a request to a server and then waits for the server to send back a reply.
- C stands for client, S for server, *idle* means “available to service requests”, *suspended* means “not available”.

# The Server as a Port Object

```
declare  
proc {ServerProc Msg}  
    case Msg  
        of calc(X Y) then  
            Y = X * X + 1.0  
    end  
end  
Server={NewAgent ServerProc}
```

- The second argument `Y` of `calc` is bound by the server.
- The server computes the polynomial `x * x + 1.0`

# What is NewAgent? (Reminder)

```
fun {NewAgent Process}
    Port Stream
in
    Port={NewPort Stream}
    thread {ForAll Stream Process} end
    Port
end
```

# The Client (using RMI)

```
declare
proc {ClientProc Msg}
  case Msg
  of work(Y) then
    Y1 Y2 in
      {Send Server calc(1.0 Y1)}
      {Wait Y1}
      {Send Server calc(2.0 Y2)}
      {Wait Y2}
    Y = Y1 + Y2
  end
end
Client={NewAgent ClientProc}
```

# The Client as a Port Object II

```
local X in  
    {Send Client work(X)}  
    {Browse X}
```

```
end
```

- Difference between the client and server:
  - The client definition references the server directly but the server definition does not know its clients.
  - The server gets a client reference indirectly, through the argument Y, i.e. the dataflow variable that is bound to the answer by the server.
  - The client waits until receiving the reply before continuing.

# What is Wait?

- `{Wait X}` suspends the thread until `X` becomes determined, i.e. also called *explicit synchronization* on variable `X`

```
declare Y
{ByNeed proc {$ X} X=1 end Y}
{Browse Y}
{Wait Y}
<statement>
```

- Display `Y` in the Browser.
- To access `Y`, the operation `{Wait Y}` will trigger the producing procedure.
- `<statement>` will be executed only after `Y` is bound

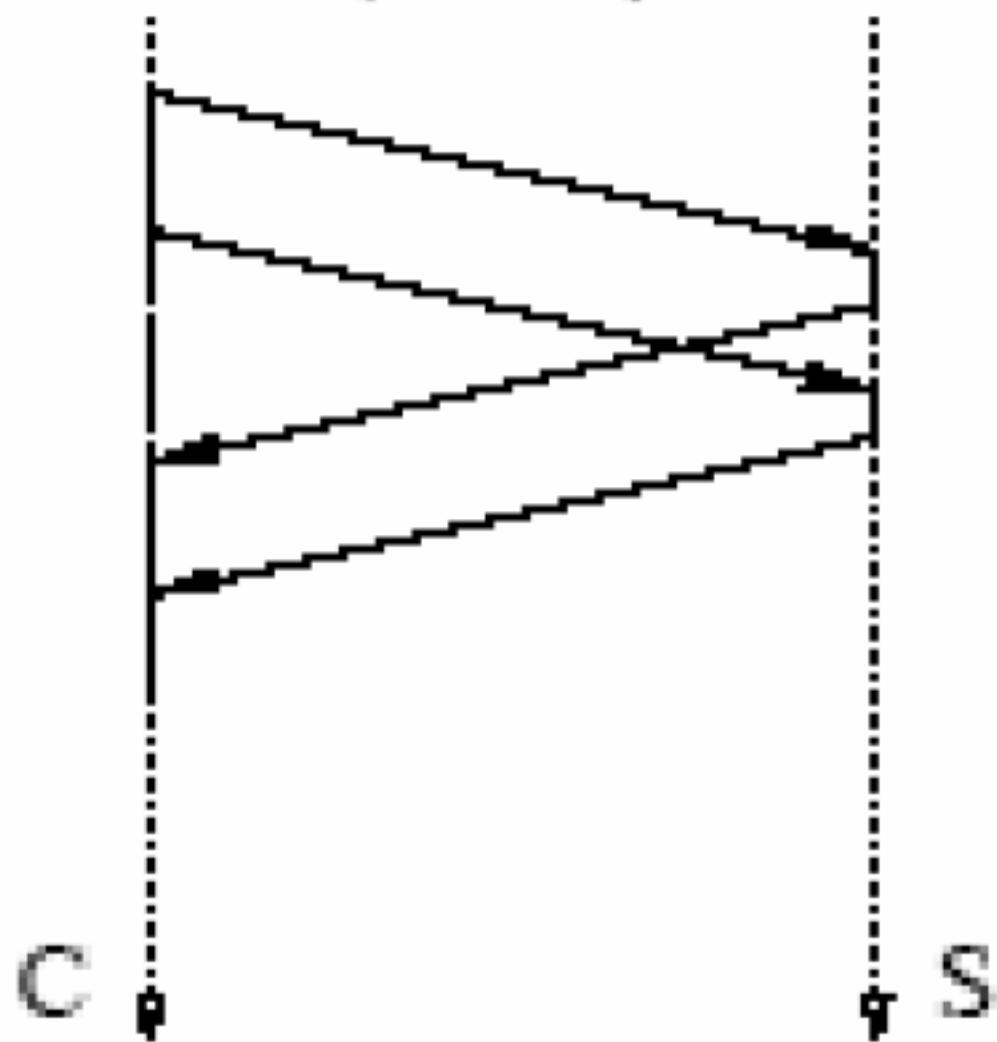
# Characteristics of RMI

- In the previous example, all messages are executed sequentially by the server.
- In practice, some RMI implementations do things somewhat differently, i.e. they allow multiple calls from different clients to be processed concurrently.
- May use different languages and different OS.

## Asynchronous RMI

- Similar to RMI, except that the client continues execution immediately after sending the request.
- The client is informed when the reply arrives.
- So, two requests can be done in rapid succession.
- **Motivation:** If communications between client and server are slow, then this will give a large performance advantage over RMI.

## 2. Asynchronous RMI *(2 calls)*



# The Asynchronous RMI Client

```
declare
proc {ClientProc Msg}
  case Msg
  of work(Y) then Y1 Y2 in
    {Send Server calc(1.0 Y1)}
    {Send Server calc(2.0 Y2)}
    Y = Y1 + Y2
  end
end
Client={NewAgent ClientProc}
local X in
  {Send Client work(X)}
  {Browse X}
end
```

# Characteristics of Asynchronous RMI

- Message sends overlap. Client waits for both results  $y_1$  and  $y_2$  before doing the addition  $y_1 + y_2$ .
- The server is the same as with standard RMI. It still receives messages one by one and executes them sequentially.
- Requests are handled by the server in the same order as they are sent and the replies arrive in that order as well.

# RMI with Callback

- Server may need to call back client to fulfill request, e.g. check on some special values.

```
proc {ServerProc Msg}
    case Msg
        of calc(X ?Y Client) then X1 D in
            {Send Client delta(D)} ← callback
            X1=X+D
            Y = X * X + 1.0
        end
    end
Server={NewAgent ServerProc}
```

# RMI with Callback

```
proc {ClientProc Msg}
    case Msg
        of work(?Z) then Y in
            {Send Server calc(10.0 Y Client)}
            Z=Y+100.0
        [] delta(?D) then D=1.0
    end
end
Client={NewAgent ClientProc}
{Browse {Send Client work($)}}
```

- Does this work?

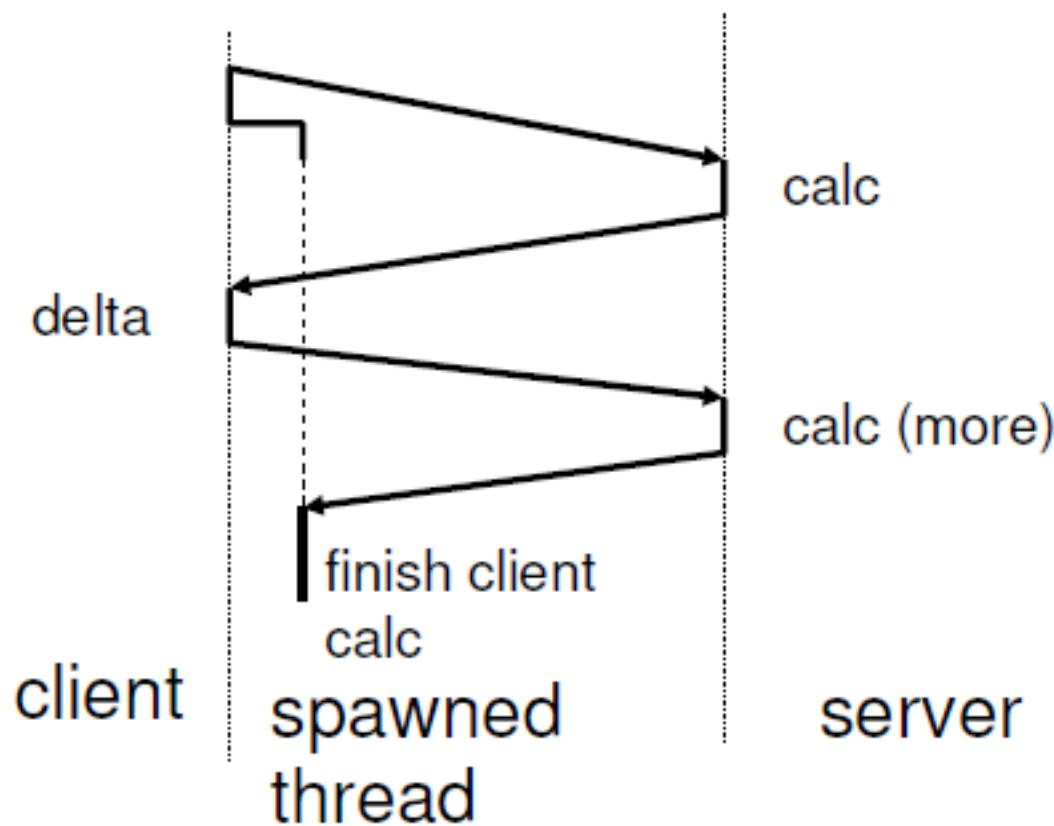
No! It deadlocks as server and client waiting for each other.

# Solution – Use Thread

```
proc {ClientProc Msg}  
  case Msg  
    of work(?Z) then Y in  
      {Send Server calc(10.0 Y Client)  
       thread Z=Y+100.0 end  
       [] delta(?D) then D=1.0  
     end  
   end
```

add thread to allow client to proceed.

# RMI with Callback (using thread)



# RMI with Callback (using continuation)

- Possible to avoid thread.

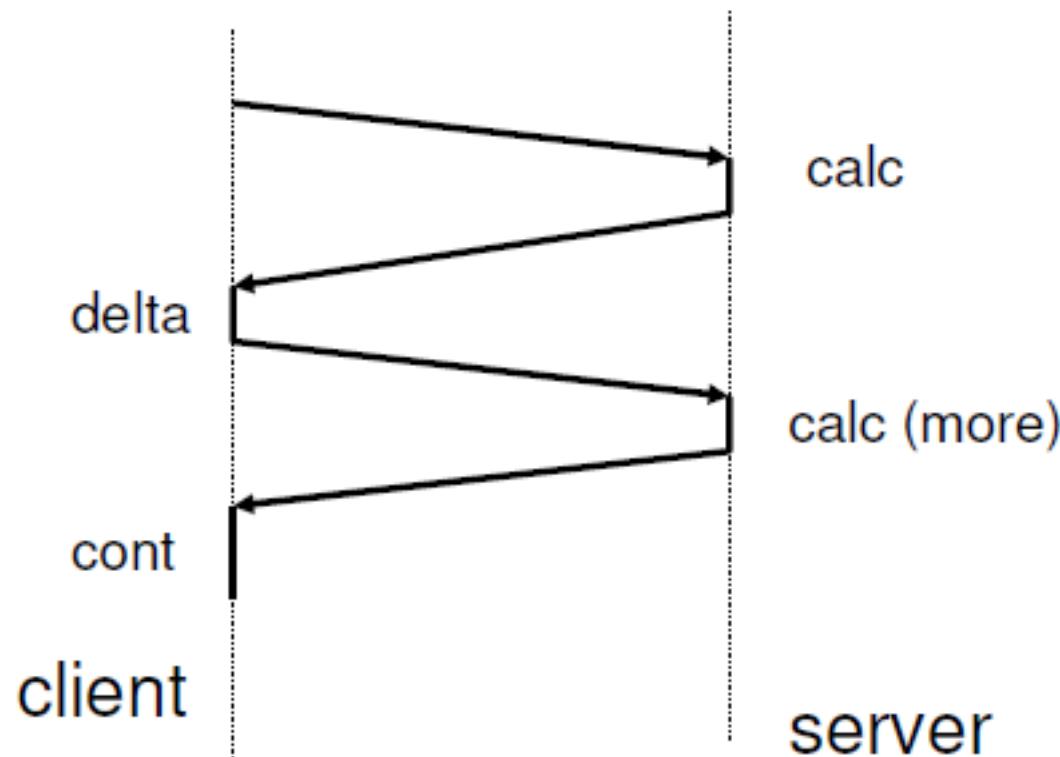
```
proc {ServerProc Msg}
  case Msg
    of calc(X Client Cont) then X1 D Y in
      {Send Client delta(D)}
      X1=X+D
      Y = X * X + 1.0
      {Send Client Cont#Y}
    end
  end
Server={NewAgent ServerProc}
```

The diagram illustrates the continuation flow in the code. A pink arrow points from the word 'Cont' in the 'calc' message to the continuation block where it is used. Another pink arrow points from the continuation block back to the original 'Cont' variable, labeled 'continuation' in blue text.

# Solution – Using Continuation Record

```
proc {ClientProc Msg}
    case Msg
        of work(?Z) then Y in
            {Send Server calc(10.0 Client cont(Z)) }
            [] cont(Z)#Y then Z=Y+100.0
            [] delta(?D) then D=1.0
    end
end
Client={NewAgent ClientProc}
{Browse {Send Client work($)} }
```

# RMI with Callback (using continuation record)



Erlang

# Erlang

- Developed by Ericsson for telecoms application.
- Features : fine grain parallelism, extreme reliability, hot code updates.
- Functional core – dynamically typed strict functional language.
- Message-passing extension – processes communicate by sending messages asynchronously in FIFO order.

# Functions in Erlang

- Uses pattern-matching and Prolog syntax

```
factorial(0) -> 1;
```

```
factorial(N) when N>0 -> N * factorial(N-1).
```

# Pattern-Matching with Tuple

```
area({square, Side}) -> Side*Side;
```

```
area({rectangle,X,Y}) -> X*Y;
```

```
area({circle, R}) -> 3.14159*R*R;
```

```
area({triangle, A,B,C}) -> ... ;
```

tuple

# Concurrency and Message Passing

- `spawn(M, F, A)` creates a new process and returns its `Pid`. Note that M-module, F-initial function, A-argument list.
- Send operation (written as `Pid!msg`) is an asynchronous message sending.
- `receive` operation removes message from a mailbox. It uses pattern-matching to select messages for removal

# An Erlang Process

```
-module(areaserver)
-export([start/0, loop/0])  
  
start() -> spawn(areaserver, loop, []).  
  
loop() -> receive  
    {From, Shape} ->  
        From!area(Shape),  
        loop()  
end.
```

spawn

receive

send

# Receive Construct

```
receive
    Pattern1 [when Guard1] -> Body1;
    :
    PatternN [when GuardN] -> BodyN;
    [after Expr -> BodyT; ]
end
```

This expression blocks until a message matching one of patterns arrives or when timeout occurs

# Summary

- Stream Object
- Thread Module and Composition
- Soft Real-Time Programming
- Agents and Message Passing
- Protocols
- Erlang