Concurrent Programming
Asynchronous message passing and Rendezvous

Verónica Gaspes
www.hh.se/staff/vero

CERES, November 19, 2007
Outline

1. Distributed programming
2. Asynchronous message passing
3. Rendezvous
Distributed programming

Asynchronous message passing

Rendezvous

Distributed Programming

Execution on distributed memory multi-computers,
Networks of machines,
Sensor networks.

Processes have their own local memory.
Exchange messages with each other!
Distributed Programming

Execution on:
- Distributed memory multicomputers,
- Networks of machines,
- Sensor networks.

Processes have their own local memory.

Exchange messages with each other!
Distributed Programming

- Execution on
  - Distributed memory multicomputers,
  - Networks of machines,
  - Sensor networks.
  - Processes have their own local memory.
  - Exchange messages with each other!

- Distributed programming
- Asynchronous message passing
- Rendezvous
Distributed Programming

Execution on
- Distributed memory multicomputers,
- Networks of machines,
- Sensor networks.

Processes have their own local memory.
Exchange messages with each other!
Distributed Programming

Execution on
- Distributed memory multicomputers,
- Networks of machines,
- Sensor networks.

Processes have their own local memory.
- Exchange messages with each other!
Distributed Programming

Execution on
- Distributed memory multicomputers,
- Networks of machines,
- Sensor networks.

- Processes have their own local memory.
- Exchange messages with each other!
Distributed Programming

What interface to the network can we expect from a programming language that supports distributed programming?

In Java the package java.net provides Socket and ServerSocket.
Distributed Programming

What **interface to the network** can we expect from a programming language that supports distributed programming?

**Read/Write**

In Java the package `java.net` provides `Socket` and `ServerSocket`.

- A socket is associated to a host machine and a port number.
- To a socket we can associate an input stream and an output stream.
- We can then read and write as to a file!
Distributed Programming

What interface to the network can we expect from a programming language that supports distributed programming?

Read/Write

In Java the package `java.net` provides `Socket` and `ServerSocket`.

- A socket is associated to a host machine and a port number.
- To a socket we can associate an input stream and an output stream.
- We can then read and write as to a file!
What **interface to the network** can we expect from a programming language that supports distributed programming?

**Read/Write**

In Java the package `java.net` provides `Socket` and `ServerSocket`.

- A socket is associated to a **host machine** and a **port number**.
- To a socket we can associate an **input stream** and an **output stream**.
- We can then read and write as to a file!
Distributed Programming

What interface to the network can we expect from a programming language that supports distributed programming?

Read/Write

In Java the package `java.net` provides `Socket` and `ServerSocket`.

- A socket is associated to a host machine and a port number.
- To a socket we can associate an input stream and an output stream.
- We can then read and write as to a file!
Distributed Programming

What interface to the network can we expect from a programming language that supports distributed programming?

Message passing primitives

- Processes share channels
- Operations using channels include some form of synchronization. Some such operations are send, receive, RPC, Rendezvous.
What **interface to the network** can we expect from a programming language that supports distributed programming?

**Message passing primitives**
- Processes share channels
- Operations using channels include some form of synchronization. Some such operations are send, receive, RPC, Rendezvous.
What interface to the network can we expect from a programming language that supports distributed programming?

Message passing primitives

- Processes share channels
- Operations using channels include some form of synchronization. Some such operations are send, receive, RPC, Rendezvous.
Distributed Programming

What interface to the network can we expect from a programming language that supports distributed programming?

Message passing primitives

- Processes share channels
- Operations using channels include some form of synchronization. Some such operations are send, receive, RPC, Rendezvous.
What **interface to the network** can we expect from a programming language that supports distributed programming?

**Message passing primitives**

- Processes share channels
- Operations using channels include some form of synchronization. Some such operations are `send`, `receive`, `RPC`, `Rendezvous`. 
Today we will study how to organize distributed programs in MPD that use *asynchronous message passing*.

**send**
Behaves very much like a V on a semaphore, but it includes a message! It is non-blocking.

**receive**
Behaves very much like a P on a semaphore, but it includes variables for the message! It is blocking.

**Channels**
*send* and *receive* are performed on a channel that the processes involved *must share*.
Send and Receive

Today we will study how to organize distributed programs in MPD that use **asynchronous message passing**.

**send**
Behaves very much like a V on a semaphore, but it includes a **message**! It is non-blocking.

**receive**
Behaves very much like a P on a semaphore, but it includes variables for the message! It is blocking.

Channels
**send** and **receive** are performed on a channel that the processes involved must share.
Today we will study how to organize distributed programs in MPD that use **asynchronous message passing**.

**send**
Behaves very much like a V on a semaphore, but it includes a **message**! It is non-blocking.

**receive**
Behaves very much like a P on a semaphore, but it includes **variables for the message**! It is blocking.

**Channels**
**send** and **receive** are performed on a channel that the processes involved **must share**.
Today we will study how to organize distributed programs in MPD that use asynchronous message passing.

**send**
Behaves very much like a $V$ on a semaphore, but it includes a **message**! It is non-blocking.

**receive**
Behaves very much like a $P$ on a semaphore, but it includes **variables for the message**! It is blocking.

**Channels**
**send** and **receive** are performed on a channel that the processes involved **must share**.
Distributed programs in MPD

But first . . . how do we write an MPD program that will run on a distributed system?

Virtual Machines

A run time environment can be created and placed on any machine you have access to (topin, regula, raynas):

```
cap vm machine = create vm()
machine = create vm() on "regula"
```

Distributed resources

A resource can be created and placed on any virtual machine:

```
cap counter c = create counter(0) on machine
cap counter d = create counter(100)
```
But first ... how do we write an MPD program that will run on a distributed system?

**Virtual Machines**

A run time environment can be created and placed on any machine you have access to (topin, regula, raynas):

```
cap vm machine = create vm()
machine = create vm() on "regula"
```

**Distributed resources**

A resource can be created and placed on any virtual machine:

```
cap counter c = create counter(0) on machine
cap counter d = create counter(100)
```
Distributed programs in MPD

But first... how do we write an MPD program that will run on a distributed system?

Virtual Machines

A run time environment can be created and placed on any machine you have access to (topin, regula, raynas):

```plaintext
cap vm machine = create vm()
machine = create vm() on "regula"
```

Distributed resources

A resource can be created and placed on any virtual machine:

```plaintext
cap counter c = create counter(0) on machine
cap counter d = create counter(100)
```
Distributed programs in MPD

But first ... how do we write an MPD program that will run on a distributed system?

Virtual Machines
A run time environment can be created and placed on any machine you have access to (topin, regula, raynas):

```mpd
cap vm machine = create vm()
machine = create vm() on "regula"
```

Distributed resources
A resource can be created and placed on any virtual machine:

```mpd
cap counter c = create counter(0) on machine
cap counter d = create counter(100)
```
Syntax and semantics

There is a type for channels, we can declare channels
\[
\text{chan name(type}_1 \text{ id}_1, \ldots, \text{ type}_n \text{ id}_n)\\
\]
for messages with datafields \( \text{id}_i \) of types \( \text{type}_i \).

A channel is
A queue of messages that have been sent but not yet received.

In MPD the type for channels is just \( \text{op} \) and the identifiers for the datafields are optional.

Example
\[
\text{op values(int)}\\
\text{op results[n](int smallest, int largest)}\\
\]
Syntax and semantics

There is a type for channels, we can declare channels

```plaintext
chan name(type₁ id₁, ..., typeₙ idₙ)
```

for messages with datafields idᵢ of types typeᵢ.

A channel is

A queue of messages that have been sent but not yet received.

In MPD the type for channels is just `op` and the identifiers for the datafields are optional.

Example

```plaintext
op values(int)
op results[n](int smallest, int largest)
```
There is a type for channels, we can declare channels

\texttt{chan name(type} \texttt{id} \texttt{, ... , type id})

for messages with datafields id of types type\texttt{.}

\textbf{A channel is}

A queue of messages that have been sent but not yet received.

\textbf{In MPD the type for channels is just op and the identifiers for the datafields are optional.}

\textbf{Example}

\texttt{op values(int)}

\texttt{op results[n](int smallest, int largest)}
There is a type for channels, we can declare channels
\[
\text{chan name}(\text{type}_1 \text{id}_1, \ldots, \text{type}_n \text{id}_n)
\]
for messages with datafields \text{id}_i of types \text{type}_i.

A channel is
A queue of messages that have been sent but not yet received.

In MPD the type for channels is just \text{op} and the identifiers for the datafields are optional.

Example
\[
\begin{align*}
\text{op} \text{ values}(\text{int}) \\
\text{op} \text{ results}[n](\text{int smallest, int largest})
\end{align*}
\]
A process sends a message to a channel `ch` by doing

```
send ch(expr_1, ..., expr_n)
```

where the types of the expressions must match the types of the message datafields in the declaration of the channel.

**Effect**

- The expressions are evaluated,
- the message containing all these values are put at the end of the queue associated with the channel,
- the process doing `send` is not delayed, the queue of messages on a channel is *unbounded*. 
A process sends a message to a channel ch by doing

\[
\text{send } \text{ch}(\text{expr}_1, \ldots, \text{expr}_n)
\]

where the types of the expressions must match the types of the message datafields in the declaration of the channel.

**Effect**

1. The expressions are evaluated,
2. the message containing all these values are put at the end of the queue associated with the channel,
3. the process doing send is not delayed, the queue of messages on a channel is unbounded.
A process sends a message to a channel \texttt{ch} by doing
\[
\text{send ch(expr}_1, \ldots, \text{expr}_n)\]
where the types of the expressions must match the types of the
message datafields in the declaration of the channel.

\textbf{Effect}

\begin{enumerate}
\item The expressions are evaluated,
\item the message containing all these values are put at the end of
the queue associated with the channel,
\item the process doing \texttt{send} is not delayed, the queue of messages
on a channel is \textit{unbounded}.
\end{enumerate}
A process sends a message to a channel `ch` by doing

\[
\text{send } \text{ch}(\text{expr}_1, \ldots, \text{expr}_n)
\]

where the types of the expressions must match the types of the message datafields in the declaration of the channel.

**Effect**

1. The expressions are evaluated,
2. the message containing all these values are put at the end of the queue associated with the channel,
3. the process doing send is not delayed, the queue of messages on a channel is *unbounded*. 
Distributed programming
Asynchronous message passing
Rendezvous

Syntax and semantics

A process **sends** a message to a channel \texttt{ch} by doing

\[
\text{send \ ch(}expr_1, \ldots, \ expr_n\text{)}
\]

where the types of the expressions must match the types of the message datafields in the declaration of the channel.

**Effect**

1. The expressions are evaluated,
2. the message containing all these values are put at the end of the queue associated with the channel,
3. the process doing \texttt{send} is not delayed, the queue of messages on a channel is \textit{unbounded}.
A process \textit{receives} a message on a channel \texttt{ch} by doing

\begin{verbatim}
receive ch(var_1, ..., var_n)
\end{verbatim}

where variables \texttt{var_i} must be of types matching the ones of the message datafields in the declaration of the channel.

\begin{itemize}
  \item Delay the receiver process until there is at least one message in the channel's queue,
  \item remove the message at the front of the queue,
  \item the fields of the message are copied to the corresponding variables.
\end{itemize}
A process receives a message on a channel `ch` by doing

\[
\text{receive } \text{ch(var}_1, \ldots, \text{var}_n)\]

where variables `var_i` must be of types matching the ones of the message datafields in the declaration of the channel.

**Effect**

1. Delay the receiver process until there is at least one message in the channel’s queue,
2. remove the message at the front of the queue,
3. the fields of the message are copied to the corresponding variables.
Syntax and semantics

A process receives a message on a channel \texttt{ch} by doing

\begin{verbatim}
receive \texttt{ch(var\textsubscript{1}, ..., var\textsubscript{n})}
\end{verbatim}

where variables \texttt{var\textsubscript{i}} must be of types matching the ones of the message datafields in the declaration of the channel.

Effect

1. \textbf{Delay} the receiver process until there is at least one message in the channel’s queue,
2. remove the message at the front of the queue,
3. the fields of the message are copied to the corresponding variables.
A process receives a message on a channel `ch` by doing

\[
\text{receive } ch(var_1, \ldots, var_n)
\]

where variables `var_i` must be of types matching the ones of the message datafields in the declaration of the channel.

**Effect**

1. **Delay** the receiver process until there is at least one message in the channel’s queue,
2. remove the message at the front of the queue,
3. the fields of the message are copied to the corresponding variables.
A process receives a message on a channel \( \text{ch} \) by doing

\[
\text{receive } \text{ch}(\text{var}_1, \ldots, \text{var}_n)
\]

where variables \( \text{var}_i \) must be of types matching the ones of the message datafields in the declaration of the channel.

**Effect**

1. **Delay** the receiver process until there is at least one message in the channel’s queue,
2. remove the message at the front of the queue,
3. the fields of the message are copied to the corresponding variables.
Syntax and semantics

Comments

- Access to the contents to each channel is atomic.
- Message delivery is reliable and error free.
- Messages are received in the order they were appended to the channel.

*If a process sends a message to a channel and later sends a second message to the same channel, the two messages will be received in the order in which they were sent.*
Distributed programming
Asynchronous message passing
Rendezvous

Syntax and semantics

Comments

- Access to the contents to each channel is atomic.
- Message delivery is reliable and error free.
- Messages are received in the order they were appended to the channel.

if a process sends a message to a channel and later sends a second message to the same channel, the two messages will be received in the order in which they were sent
Syntax and semantics

Comments

- Access to the contents to each channel is atomic.
- Message delivery is reliable and error free.
- Messages are received in the order they were appended to the channel.

if a process sends a message to a channel and later sends a second message to the same channel, the two messages will be received in the order in which they were sent
Syntax and semantics

Comments

- Access to the contents to each channel is atomic.
- Message delivery is reliable and error free.
- Messages are received in the order they were appended to the channel.

If a process sends a message to a channel and later sends a second message to the same channel, the two messages will be received in the order in which they were sent.
Comments

- Access to the contents to each channel is atomic.
- Message delivery is reliable and error free.
- Messages are received in the order they were appended to the channel.

*if a process sends a message to a channel and later sends a second message to the same channel, the two messages will be received in the order in which they were sent*
We want to explore some communication patterns and understand the tradeoffs between them. These patterns occur often in distributed parallel computations and in decentralized distributed systems.

Example

Assume there are $n$ processes and that each process knows a local integer value $v$. The goal is for every process to learn the smallest and largest of all the local values.
We want to explore some communication patterns and understand the tradeoffs between them. These patterns occur often in distributed parallel computations and in decentralized distributed systems.

**Example**

Assume there are \( n \) processes and that each process knows a local integer value \( v \). The goal is for every process to learn the smallest and largest of all the local values.
A centralized solution

One process, \( P_0 \),
- collects all values,
- calculates the largest and the smallest,
- broadcasts the results to all other processes.

All other processes \( P_1, P_2, \ldots, P_{n-1} \)
send their values to \( P_0 \) and wait to be sent the result!
A centralized solution

One process, $P_0$,
- collects all values,
- calculates the largest and the smallest,
- broadcasts the results to all other processes.

All other processes $P_1, P_2, \ldots, P_{n-1}$ send their values to $P_0$ and wait to be sent the result!
A centralized solution

One process, $P_0$,
1. collects all values,
2. calculates the largest and the smallest,
3. broadcasts the results to all other processes.

All other processes $P_1, P_2, \ldots, P_{n-1}$ send their values to $P_0$ and wait to be sent the result!
A centralized solution

One process, $P_0$,

1. collects all values,
2. calculates the largest and the smallest,
3. broadcasts the results to all other processes.

All other processes $P_1, P_2, \ldots, P_{n-1}$ send their values to $P_0$ and wait to be sent the result!
A centralized solution

One process, $P_0$,
1. collects all values,
2. calculates the largest and the smallest,
3. broadcasts the results to all other processes.

All other processes $P_1, P_2, \ldots, P_{n-1}$ send their values to $P_0$ and wait to be sent the result!
A centralized solution

One process, $P_0$,
1. collects all values,
2. calculates the largest and the smallest,
3. broadcasts the results to all other processes.

All other processes $P_1, P_2, \ldots, P_{n-1}$ send their values to $P_0$ and wait to be sent the result!
resource minMax1()
    int n; getarg(1,n)
    op values(int)
    op results[n-1](int smallest, int largest)

process T0{...}

process T[i = 1 to n-1]{
    int v = int(floor(random()*100));
    int smallest, largest
    send values(v)
    receive results[i](smallest,largest)
    write(i,"reporting",v,smallest,largest)
}
end
A centralized solution

```plaintext
process T0{
    int v = int(floor(random() * 100))
    int tmp, smallest = v, largest = v

    for [i = 1 to n-1] {
        receive values(tmp)
        if (tmp < smallest) smallest = tmp
        if (tmp > largest) largest = tmp
    }

    for [i = 1 to n-1] {
        send results[i](smallest, largest)
    }

    write(0, "reporting", v, smallest, largest)
}
```
A symmetric solution

Each process $P_i$ executes the same algorithm:
- send the local value to all other processes
- calculate the largest and the smallest!
A symmetric solution

Each process $P_i$ executes the same algorithm:

1. send the local value to all other processes
2. calculate the largest and the smallest
A symmetric solution

Each process $P_i$ executes the same algorithm:

1. send the local value to all other processes
2. calculate the largest and the smallest!
A symmetric solution

```
op values[0:n-1](int)

process T[i = 0 to n-1]{
    int v = int(floor(random() * 100));
    int tmp, smallest = v, largest = v
    for[j=0 to n-1 st j!=i]{
        send values[j](v)
    }
    for[j=1 to n-1]{
        receive values[i](tmp)
        if(tmp < smallest) smallest = tmp
        if(tmp > largest) largest = tmp
    }
    write(i,"reporting",v,smallest,largest)
}
```
A circular ring

Organize the processes in a circular ring where each process sends to its successor and receives from its predecessor.

Execution in 2 stages

1. **Stage 1:**
   1. Receive values from predecessor,
   2. Calculate smallest and largest with local value,
   3. Send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. Send values to the next.
A circular ring

Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

Execution in 2 stages:

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
A circular ring

Organize the processes in a circular ring where each process sends to its successor and receives from its predecessor.

Execution in 2 stages

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
A circular ring

Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

**Execution in 2 stages**

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
Distributed programming
Asynchronous message passing
Rendezvous

A circular ring

Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

Execution in 2 stages

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
**A circular ring**

Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

**Execution in 2 stages**

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
A circular ring

Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

**Execution in 2 stages**

1. **Stage 1:**
   - 1. Receive values from predecessor,
   - 2. calculate smallest and largest with local value,
   - 3. send values to successor.

2. **Stage 2:**
   - 1. Receive the global smallest and largest,
   - 2. send them to the successor.
Organize the processes in a **circular ring** where each process sends to its successor and receives from its predecessor.

**Execution in 2 stages**

1. **Stage 1:**
   1. Receive values from predecessor,
   2. calculate smallest and largest with local value,
   3. send values to successor.

2. **Stage 2:**
   1. Receive the global smallest and largest,
   2. send them to the successor.
Distributed programming

Asynchronous message passing

Rendezvous

A circular ring

```plaintext
op values[0:n-1](int smallest, int largest)

process T0{ ... }

process T[i = 1 to n-1]{
    int v = int(floor(random()*100))
    int smallest, largest

    receive values[i](smallest, largest)
    if(v < smallest) smallest = v  
    if(v > largest) largest = v

    send values[(i+1) mod n](smallest, largest)

    receive values[i](smallest, largest)

    if(i<n-1){ send values[i+1](smallest, largest) }

    write(i,"reporting",v,smallest,largest)
}
end
```
A circular ring

process T0{
    int v = int(floor(random() * 100))
    int smallest = v, largest = v

    send values[1](smallest, largest)
    receive values[0](smallest, largest)
    send values[1](smallest, largest)

    write(0, "reporting", v, smallest, largest)
}

Remarks

- Distributed programming
- Asynchronous message passing
- Rendezvous

The queue of $P_0$ grows rapidly.

- Easiest to program!
- $n(n-1)$ messages
- Communication overhead can reduce performance.

- $2(n-1)$ messages
- $P_n - 1$ has to wait a lot!
Remarks

- $2(n-1)$ messages.
- The queue of $P_0$ grows rapidly.
- Easiest to program!
Remarks

- Easiest to program!
- \(2(n-1)\) messages.
- \(n(n-1)\) messages.
- Communication overhead can reduce performance.
- \(P_0\) has to wait a lot!

- The queue of \(P_0\) grows rapidly.

- \(2(n-1)\) messages.
Remarks

- 2(n-1) messages.
- The queue of \( P_0 \) grows rapidly.
- Easiest to program!
- \( n(n-1) \) messages.
- Communication overhead can reduce performance.
- \( 2(n-1) \) messages
- \( P_{n-1} \) has to wait a lot!
Remarks

- 2(n-1) messages.
- The queue of $P_0$ grows rapidly.

- Easiest to program!
- $n(n-1)$ messages.
- Communication overhead can reduce performance.
- $2(n-1)$ messages
- $P_{n-1}$ has to wait a lot!
Remarks

- $2(n-1)$ messages.
- The queue of $P_0$ grows rapidly.
- Easiest to program!
  - $n(n-1)$ messages.
  - Communication overhead can reduce performance.
- $2(n-1)$ messages
- $P_{n-1}$ has to wait a lot!
Remarks

- 2(n-1) messages.
- The queue of P₀ grows rapidly.
- Easiest to program!
- n(n-1) messages.
- Communication overhead can reduce performance.
- 2(n-1) messages
- Pₙ₋₁ has to wait a lot!
Remarks

- $2(n-1)$ messages.
- The queue of $P_0$ grows rapidly.
- Easiest to program!
- $n(n-1)$ messages.
- Communication overhead can reduce performance.

- $P_{n-1}$ has to wait a lot!
Remarks

- \(2(n-1)\) messages.
- The queue of \(P_0\) grows rapidly.

- Easiest to program!
- \(n(n-1)\) messages.
- Communication overhead can reduce performance.

- \(2(n-1)\) messages
- \(P_{n-1}\) has to wait a lot!
Remarks

- $2(n-1)$ messages.
- The queue of $P_0$ grows rapidly.

- Easiest to program!
- $n(n-1)$ messages.
- Communication overhead can reduce performance.

- $2(n-1)$ messages
- $P_{n-1}$ has to wait a lot!
Remarks

- 2(n-1) messages.
- The queue of P₀ grows rapidly.

Easiest to program!
- n(n-1) messages.
- Communication overhead can reduce performance.

- 2(n-1) messages
- Pₙ₋₁ has to wait a lot!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include

- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include
- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process leaving the sender process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include:

- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process leaving the sender process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include:

- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process leaving the sender process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include

- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
It is very difficult to synchronize processes using asynchronous message passing! We would like to attack problems like waiting on conditions in the setting of distributed programming.

Think of the primitive of sending a message to another process leaving the sender process suspended until the message is received and processed! At that point the processes can handshake or rendezvous.

The receiving operation will have a queue of waiting processes, instead of a queue of messages!

Implementations of these primitives in programming languages can include

- Returning a value to the calling process!
- Add conditions to the receive primitive for blocking senders!
The channel is declared as an op with fields for the message and possibly a return type:

```c
resource TimeServer
    op getTime() returns int
    op delay(int)
```
The operation is serviced by an in statement:

```
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!
The operation is *serviced* by an `in` statement:

```plaintext
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!

- Between `in` and `->`: the guard
- Between `in` and `ni`: a guarded operation
Rendezvous in MPD

The operation is *serviced* by an **in** statement:

```plaintext
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!

- **in** and `->`: the guard
- **in** and **ni**: a guarded operation
Rendezvous in MPD

The operation is *serviced* by an `in` statement:

```
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!

- Between `in` and `->`: the guard
- Between `in` and `ni`: a guarded operation

- The process executing the `in` is delayed until there is at least one pending call on the channel `getTime`
- Then it selects the oldest pending call and executes the operation `time = tod`
- At this point both processes can proceed!
The operation is *serviced* by an `in` statement:

\[
in \text{getTime()} \text{ returns } \text{time} \rightarrow \text{time} = \text{tod ni}
\]
in a process that is executing concurrently with the sender!

- Between `in` and `->`: the guard
- Between `in` and `ni`: a guarded operation

The process executing the `in` is delayed until there is at least one pending call on the *channel* `getTime`

- Then it selects the oldest pending call and executes the operation `time = tod`
- At this point both processes can proceed!
Rendezvous in MPD

The operation is serviced by an **in** statement:

```
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!

- Between **in** and **->**: the guard
- Between **in** and **ni**: a guarded operation

- The process executing the **in** is delayed until there is at least one pending call on the channel **getTime**
- Then it selects the oldest pending call and executes the operation **time = tod**
- At this point both processes can proceed!
The operation is *serviced* by an `in` statement:

```
in getTime() returns time -> time = tod ni
```

in a process that is executing concurrently with the sender!

- Between `in` and `->`: the guard
- Between `in` and `ni`: a guarded operation

- The process executing the `in` is delayed until there is at least one pending call on the channel `getTime`
- Then it selects the oldest pending call and executes the operation `time = tod`
- At this point both processes can proceed!
Rendezvous in MPD

The general form of \texttt{in}

A process servicing an \texttt{in} can serve several alternatives:

\begin{verbatim}
\texttt{in} getTime() \texttt{returns} time \quad \rightarrow \quad \texttt{time} = \texttt{tod}
\texttt{[] delay(waketime) \texttt{and} waketime} \leq \texttt{tod} \quad \rightarrow \quad \texttt{skip}
\texttt{ni}
\end{verbatim}

and a guard can include a boolean expression!

The process executing the \texttt{in} will

- Delay until some guard succeeds (there is a call to the channel and the boolean expression is true).
- The oldest invocation is served by executing its action.
- After, both the caller and the service can continue.
Rendezvous in MPD

The general form of in

A process servicing an in can serve several alternatives:

\[ \text{in} \ \text{getTime()} \ \text{returns} \ \text{time} \rightarrow \text{time} = \text{tod} \]
\[ \text{[] delay(waketime) and waketime} \leq \text{tod} \rightarrow \text{skip} \]
\[ \text{ni} \]

and a guard can include a boolean expression!

The process executing the in will

- Delay until some guard succeeds (there is a call to the channel and the boolean expression is true).
- The oldest invocation is served by executing its action.
- After, both the caller and the service can continue.
The general form of `in`

A process servicing an `in` can serve several alternatives:

```
in getTime() returns time  ->  time = tod
[] delay(waketime) and waketime <= tod  ->  skip
ni
```

and a guard can include a boolean expression!

The process executing the `in` will

- Delay until some guard succeeds (there is a call to the channel and the boolean expression is true).
- The oldest invocation is served by executing its action.
- After, both the caller and the service can continue.
The general form of `in`:

A process servicing an `in` can serve several alternatives:

```
in getTime() returns time  ->  time = tod
[] delay(waketime) and waketime <= tod  ->  skip
ni
```

and a guard can include a boolean expression!

The process executing the `in` will:

- Delay until some guard succeeds (there is a call to the channel and the boolean expression is true).
- The oldest invocation is served by executing its action.
- After, both the caller and the service can continue.
Rendezvous in MPD

The general form of \texttt{in}

A process servicing an \texttt{in} can serve several alternatives:

\texttt{in} \quad \texttt{getTime()} \quad \texttt{return} \quad \texttt{time} \quad \rightarrow \quad \texttt{time} = \texttt{tod}

[] \quad \texttt{delay(waketime)} \quad \texttt{and} \quad \texttt{waketime} \leq \texttt{tod} \quad \rightarrow \quad \texttt{skip}

\texttt{ni}

and a guard can include a boolean expression!

The process executing the \texttt{in} will

- Delay until some guard succeeds (there is a call to the channel and the boolean expression is true).
- The oldest invocation is served by executing its action.
- After, both the caller and the service can continue.
A time server

A process that can be used by other processes to be delayed until some deadline. Processes can use the time server to set up alarms!
A time server

A process that can be used by other processes to be delayed until some deadline. Processes can use the time server to set up alarms!
A time server

A process that can be used by other processes to be delayed until some deadline. Processes can use the time server to set up alarms!
A time server

A process that can be used by other processes to be delayed until some deadline. Processes can use the time server to set up alarms!
The time server

```plaintext
resource TimeServer
  op getTime() returns int
  op delay(int)
body TimeServer()
process timer{
  int tod = 0     # time of day
  while(true){
    in getTime() returns time -> time = tod
    [] delay(waketime) and waketime <= tod -> skip ni
  }
}
end
```
Using the time server

Example

resource Test
  import TimeServer
body Test()
  cap TimeServer ts = create TimeServer()
process A{
  int i = 0;
  while(true){
    call ts.delay(20*i++);
    write(ts.getTime()/20);
  }
}
Using the time server

Example

```plaintext
resource Test

import TimeServer

body Test()

cap TimeServer ts = create TimeServer()

process A{
    int i = 0;
    while(true){
        call ts.delay(20*i++);
        write(ts.getTime()/20);
    }
}
```

Notice the use of `call`! When we use the results we can omit the call (as inside the `write`).
Asynchronous message passing revisited

Example

How did the TimeServer keep track of the time? (increment \texttt{tod})

In MPD a receive is just an abbreviation for an operation serviced by an \texttt{rv} and invoked with a \texttt{send}!
Asynchronous message passing revisited

Example

How did the **TimeServer** keep track of the time? (increment **tod**)

- There will be a hardware process associated to the computer clock that **asynchronously** and periodically sends messages to the **TimeServer**!
- On receiving such messages **tod** will be incremented.

In MPD a receive is just an abbreviation for an operation serviced by an **in** and invoked with a **send**!
Asynchronous message passing revisited

Example

How did the **TimeServer** keep track of the time? (increment **tod**)

- There will be a hardware process associated to the computer clock that **asynchronously** and periodically sends messages to the **TimeServer**!
- On receiving such messages **tod** will be incremented.
Asynchronous message passing revisited

Example

How did the TimeServer keep track of the time? (increment tod)

- There will be a hardware process associated to the computer clock that asynchronously and periodically sends messages to the TimeServer!
- On receiving such messages tod will be incremented.

In MPD a receive is just an abbreviation for an operation serviced by an in and invoked with a send!

- Serviced by in invoked by call: Rendezvous!
- Serviced by in invoked by send: Asynchronous message passing!
Asynchronous message passing revisited

Example

How did the TimeServer keep track of the time? (increment \texttt{tod})

- There will be a hardware process associated to the computer clock that asynchronously and periodically sends messages to the TimeServer!
- On receiving such messages \texttt{tod} will be incremented.

In MPD a receive is just an abbreviation for an operation serviced by an \texttt{in} and invoked with a \texttt{send}!

- Serviced by \texttt{in} invoked by \texttt{call}: Rendezvous!
- Serviced by \texttt{in} invoked by \texttt{send}: Asynchronous message passing!
Asynchronous message passing revisited

Example

How did the TimeServer keep track of the time? (increment \textit{tod})

- There will be a hardware process associated to the computer clock that \textit{asynchronously} and periodically sends messages to the TimeServer!
- On receiving such messages \textit{tod} will be incremented.

In MPD a receive is just an abbreviation for an operation serviced by an \textit{in} and invoked with a \textit{send}!
- Serviced by \textit{in} invoked by \textit{call}: Rendezvous!
- Serviced by \textit{in} invoked by \textit{send}: Asynchronous message passing!
The time server revisited

```plaintext
resource TimeServer
  op getTime() returns int
  op delay(int)
  op tick()
body TimeServer()
  process timer{
    int tod = 0  # time of day
    while(true){
      in getTime() returns time -> time = tod
      [] delay(waketime) and waketime <= tod -> skip
      [] tick() -> tod++
      ni
    }
  }
end
```