Concurrent Programming
Distributed programming - asynchronous message passing

1. Monitors: where are the threads?
2. Distributed programming
3. Asynchronous message passing

November 22
www.hh.se/staff/vero/concurrent
We said that monitors are abstract data types with support for:

1. mutual exclusion
2. waiting on conditions

But, where are the threads?

We have to spend some time thinking how to organize our programs.
Monitors

We said that monitors are abstract data types with support for

1. mutual exclusion
2. waiting on conditions

But, where are the threads?

We have to spend some time thinking how to organize our programs.
We said that monitors are abstract data types with support for
1. mutual exclusion
2. waiting on conditions

But, where are the threads?

We have to spend some time thinking how to organize our programs.
An object will be shared by many threads ...

```java
Example

class BoundedBuffer<T>{
    private int n;
    private T[] buf;
    private int front = 0;
    private int rear = 0;
    /* constructor */
    public BoundedBuffer(int size)
    /* operations */
    public void deposit(T item){
        buf[rear] = item;
        rear = inc(rear);
    }
    public T fetch(){...}
}
```
Monitors: where are the threads?

Distributed programming

Asynchronous message passing

Make it thread safe (make it a monitor!)

Example

```java
import java.util.concurrent.locks.*;

class BoundedBuffer<T> {
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();
    private int count = 0;
    public void deposit(T item) {
        lock.lock();
        try{
            while (count == n) notFull.await();
            buf[rear] = item;
            rear = inc(rear);
            count++;
            notEmpty.signal();
        }catch(...){...}
        finally {lock.unlock();
    }
}
```
Now many threads can use it concurrently ...

Example

```java
public static void main(String[] args){
    BoundedBuffer<Integer> theBuffer =
        new BoundedBuffer<Integer>(5);
    for(int i = 1;i<10;i++){
        new Producer(theBuffer, i).start();
    }
    for(int i = 1;i<5;i++){
        new Consumer(theBuffer, i).start();
    }
}
```
Now many threads can use it concurrently . . .

Example

class Producer extends Thread{
    private BoundedBuffer<Integer> bb;
    private int id;
    private Random random = new Random();
    public Producer(BoundedBuffer<Integer> bb, int id){
        this.bb = bb;this.id = id;
    }
    public void run(){
        int x;
        while(true){
            nap(random.nextInt(5000));
            x = random.nextInt(100);
            bb.deposit(x);
        }
    }
}
What happens to the threads?

In a concurrent program, many threads will be making progress simultaneously.

If many threads share an instance of BoundedBuffer, only one will get to execute fetch or deposit at a time! (they appear to be atomic!)

And the other threads?
They are put to sleep waiting for permission to run the method they want!

There is also an explicit way of putting threads to sleep, namely the method await on a condition variable!

And a way of explicitly waking a thread, namely the method signal on a condition variable.

Who does a signal?
Another thread that when running discovers that it has made some condition valid!
What happens to the threads?

In a concurrent program, many threads will be making progress simultaneously.

If many threads share an instance of BoundedBuffer, only one will get to execute fetch or deposit at a time! (they appear to be atomic!)

And the other threads? They are put to sleep waiting for permission to run the method they want!

There is also an explicit way of putting threads to sleep, namely the method await on a condition variable!

And a way of explicitly waking a thread, namely the method signal on a condition variable.

Who does a signal? Another thread that when running discovers that it has made some condition valid!
What happens to the threads?

In a concurrent program, many threads will be making progress simultaneously. If many threads share an instance of BoundedBuffer, only one will get to execute `fetch` or `deposit` at a time! (they appear to be atomic!)

And the other threads? They are put to `sleep` waiting for permission to run the method they want!

There is also an explicit way of putting threads to sleep, namely the method `await` on a condition variable!

And a way of explicitly waking a thread, namely the method `signal` on a condition variable.

Who does a signal? Another thread that when running discovers that it has made some condition valid!
In a concurrent program, many threads will be making progress simultaneously.

If many threads share an instance of BoundedBuffer, only one will get to execute fetch or deposit at a time! (they appear to be atomic!)

And the other threads?
They are put to sleep waiting for permission to run the method they want!

There is also an explicit way of putting threads to sleep, namely the method await on a condition variable!

And a way of explicitly waking a thread, namely the method signal on a condition variable.

Who does a signal?
Another thread that when running discovers that it has made some condition valid!
What happens to the threads?

In a concurrent program, many threads will be making progress simultaneously.

If many threads share an instance of BoundedBuffer, only one will get to execute `fetch` or `deposit` at a time! (they appear to be atomic!)

There is also an explicit way of putting threads to `sleep`, namely the method `await` on a condition variable!

And a way of explicitly waking a thread, namely the method `signal` on a condition variable.

Who does a signal?

Another thread that when running discovers that it has made some condition valid!

And the other threads?

They are put to `sleep` waiting for permission to run the method they want!
What happens to the threads?

In a concurrent program, many threads will be making progress simultaneously.

If many threads share an instance of BoundedBuffer, only one will get to execute fetch or deposit at a time! (they appear to be atomic!)

And the other threads? They are put to sleep waiting for permission to run the method they want!

There is also an explicit way of putting threads to sleep, namely the method await on a condition variable!

And a way of explicitly waking a thread, namely the method signal on a condition variable.

Who does a signal? Another thread that when running discovers that it has made some condition valid!
How can we use monitors to schedule a shared object?

We now assume that we have a shared object (that we cannot re-program!) but that needs to be shared by concurrent threads.

Can we use monitors to make it thread safe?

**Example**

A database that is shared by *reader* threads and *writer* threads. We assume there are methods already available to read and write to the database!

**Interpose** a scheduler that the concurrent threads must use

<table>
<thead>
<tr>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestRead</td>
<td>requestWrite</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>releaseRead</td>
<td>releaseWrite</td>
</tr>
</tbody>
</table>
How can we use monitors to *schedule* a shared object?

We now assume that we have a shared object (that we cannot re-program!) but that needs to be shared by concurrent threads.

Can we use monitors to make it *thread safe*?

**Example**

A database that is shared by *reader* threads and *writer* threads. We assume there are methods already available to read and write to the database!

*Interpose* a scheduler that the concurrent threads *must* use

<table>
<thead>
<tr>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestRead</td>
<td>requestWrite</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>releaseRead</td>
<td>releaseWrite</td>
</tr>
</tbody>
</table>
How can we use monitors to *schedule* a shared object?

We now assume that we have a shared object (that we cannot re-program!) but that needs to be shared by concurrent threads.

Can we use monitors to make it *thread safe*?

**Example**

A database that is shared by *reader* threads and *writer* threads. We assume there are methods already available to read and write to the database!

**Interpose** a scheduler that the concurrent threads *must* use

<table>
<thead>
<tr>
<th>reader</th>
<th>writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestRead</td>
<td>requestWrite</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>releaseRead</td>
<td>releaseWrite</td>
</tr>
</tbody>
</table>
What does the scheduler need to know?

It has to know what policy to implement.

Example

The database scheduler has to guarantee that

- Processes that request to read while there is a writer writing are blocked
- Processes that request to write while there is a writer writing or readers reading are blocked
- Processes that are allowed to proceed after a request will be able to do what they have to do safely
- Processes that do release should try to wake up processes that are waiting!
What does the scheduler need to know?

It has to know **what policy to implement**.

**Example**

The database scheduler has to guarantee that

- Processes that request to read while **there is a writer writing** are blocked
- Processes that request to write while **there is a writer writing** or readers reading are blocked
- Processes that are allowed to proceed after a request will be able to do what they have to do safely
- Processes that do release should try to wake up processes that are waiting!
What does the scheduler need to know?

It has to know **what policy to implement**.

**Example**

The database scheduler has to guarantee that

- Processes that request to read while **there is a writer writing** are blocked
- Processes that request to write while **there is a writer writing** or readers reading are blocked
- Processes that are allowed to proceed after a request will be able to do what they have to do safely
- Processes that do release should try to wake up processes that are waiting!
What does the scheduler need to know?

It has to know **what policy to implement**.

**Example**

The database scheduler has to guarantee that

- Processes that request to read while **there is a writer writing** are blocked
- Processes that request to write while **there is a writer writing or readers reading** are blocked
- Processes that are allowed to proceed after a request will be able to do what they have to do safely
- Processes that do release should try to wake up processes that are waiting!
It has to know what policy to implement.

**Example**

The database scheduler has to guarantee that:

- Processes that request to read while there is a writer writing are blocked.
- Processes that request to write while there is a writer writing or readers reading are blocked.
- Processes that are allowed to proceed after a request will be able to do what they have to do safely.
- Processes that do release should try to wake up processes that are waiting!
What does the scheduler need to know?

It has to know **what policy to implement**.

**Example**

The database scheduler has to guarantee that

- Processes that request to read while **there is a writer writing** are blocked
- Processes that request to write while **there is a writer writing or readers reading** are blocked
- Processes that are allowed to proceed after a request will be able to do what they have to do safely
- Processes that do release should try to wake up processes that are waiting!
A database

Example

class DB {
    public void write(int id) {
        nap(id*500);
        System.out.println("writer" + id + " is writing");
        nap(id*500);
    }
    public void read(int id) {
        nap(id*500);
        System.out.println("reader" + id + " is reading");
        nap(id*500);
    }
}
The threads

class Writer extends Thread{
    private RWScheduler rws;
    private DB db;

    public void run(){
        while(true){
            rws.requestWrite();
            System.out.println("writer"+id+ " entering");
            db.write(id);
            rws.releaseWrite();
            System.out.println("writer"+id+ " leaving");
        }
    }
}
The threads

Example

class Reader extends Thread{
    private RWScheduler rws;
    private DB db;

    public void run(){
        while(true){
            rws.requestRead();
            System.out.println("reader"+id+ " entering");
            db.read(id);
            rws.releaseRead();
            System.out.println("reader"+id+ " leaving");
        }
    }
}
The scheduler (a monitor!)

Example

```java
import java.util.concurrent.locks.*;

class RWScheduler{
    private int nr = 0;
    private int nw = 0;
    private int dr = 0;
    private int dw = 0;

    private Lock lock = new ReentrantLock();
    private Condition reader = lock.newCondition();
    private Condition writer = lock.newCondition();

    public void requestRead()
    public void requestWrite()
    public void releaseRead()
    public void releaseWrite()
}
```
The scheduler (a monitor!)

Example

```java
import java.util.concurrent.locks.*;
class RWScheduler{
    public void requestRead(){
        lock.lock();
        try{
            while(nw > 0) dr++;reader.await();
            dr--;nr++;
            if(nr==1)reader.signalAll();
        }catch(InterruptedException e){
        }
    }finally{lock.unlock();}
}
```
The scheduler (a monitor!)

Example

```java
import java.util.concurrent.locks.*;
class RWScheduler{
    public void releaseRead(){
        lock.lock();
        try{
            nr--;
            if(nr==0 && dw>0){writer.signal();}
        }finally{lock.unlock();}
    }
}
```
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

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

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

- 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?

**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!
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 \texttt{java.net} provides \texttt{Socket} and \texttt{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.
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`.
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`.
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.
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**

\( \text{send and receive} \) 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**.
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.
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()
```

```cap
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):

```
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*):

```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)
```
There is a type for channels, we can declare channels

```
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

```
op values(int)
op results[n](int smallest, int largest)
```
Syntax and semantics

There is a type for channels, we can declare channels

\[
\text{\texttt{chan name}(type}_1 \ \text{id}_1, \ldots, \ \text{type}_n \ \text{id}_n)\
\]

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 \texttt{op} and the identifiers for the datafields are optional.

Example

\[
\text{\texttt{op values}(int)}
\]

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

There is a type for channels, we can declare channels

```
chan name(type_1 id_1, ..., type_n id_n)
```

for messages with datafields id; of types type;.

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

```
op values(int)
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

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

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

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 \texttt{send} is not delayed, the queue of messages on a channel is \textit{unbounded}.
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**

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 `ch` by doing

```
send ch(expr₁, ..., exprₙ)
```

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 \( \text{ch} \) 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

\begin{verbatim}
send ch(expr_1, ..., expr_n)
\end{verbatim}

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 receives a message on a channel $\text{ch}$ by doing

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

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

**Effect**

- Delay the receiver process until there is at least one message in the channel's queue,
- Remove the message at the front of the queue,
- The fields of the message are copied to the corresponding variables.
A process **receives** a message on a channel `ch` by doing

```
  receive ch(var₁, ..., varₙ)
```

where variables `varᵢ` 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 $ch$ by doing

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

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.
A process receives a message on a channel `ch` by doing

```
receive ch(var₁, ..., varₙ)
```

where variables `varᵢ` 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 `ch` by doing

```
receive ch(var₁, ..., varₙ)
```

where variables `varᵢ` 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.
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.
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
We want to explore some communication patterns and understand the tradeoffs between them. These patterns occurr often in distributed parallel computations and in descentralized 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, $P_0$,

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!
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

```plaintext
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:

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

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

```plaintext
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

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

- **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

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

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.
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.
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

```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

```java
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

- $2(n-1)$ 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 $P_0$ 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!