Haute école d’ingénierie et d’architecture Fribourg
Hochschule für Technik und Architektur Freiburg

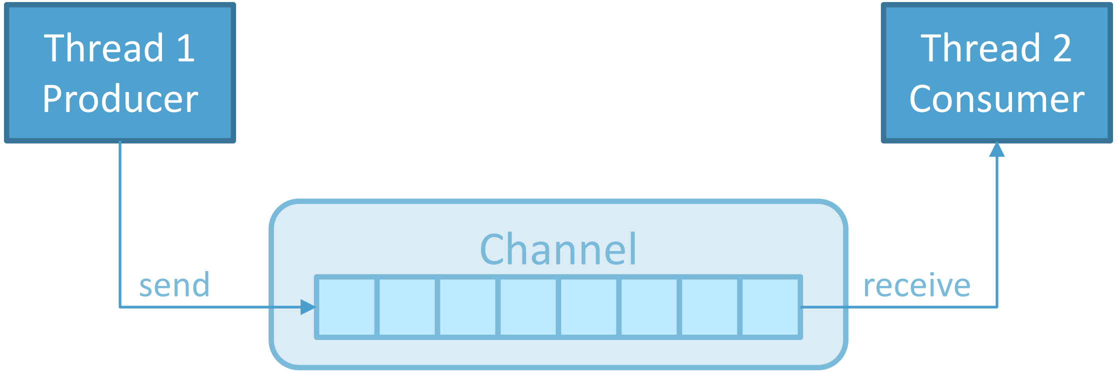
Asynchronous Message Passing

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

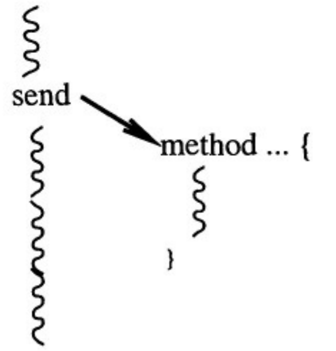
Asynchronous Message Passing

JR book chapter 7, https://en.wikipedia.org/wiki/Message_passing

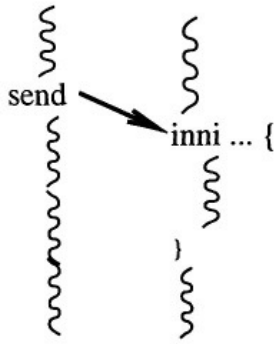
- Processes share **channels** …
  - Abstraction of a physical communication network
- … to communicate between processes
  - **`send`** and **`receive`**



Async Message Passing in JR



dynamic process creation



asynchronous message passing

Serviced by: **method (`op`)** **input statement (`inni`)**

Synchronization (1/2)

- Communication is accomplished since data flows from the sender to the receiver

- Synchronization is accomplished since a message **cannot be read before it is send**

Synchronization (2/2)

- When message passing is used:
  → channels are the only object that are shared between processes
- Absence of shared variables:
  → No shared memory used
- Due to the absence of shared variables:
  → Processes can be distributed among processors

Distributed programs

Distributed programs

- Can run on distributed processors

- Can run on distributed computers/VMs
  - Some additional effort must be done here, we'll see later

- But can also run on in a single processor environment, just as any other concurrent program

Channels

- Can be global to processes
- Can be connected to a single sender and receiver
- Can be connected to multiple senders and receivers

- Dataflow can be
  - unidirectional
  - bidirectional

- Communication can be synchronous or asynchronous

Different kind of processes

- Basically there are 4 kinds of processes in a distributed program:
  - Clients
  - Servers
  - Filters
  - Peers

Different kind of processes

- Basically there are 4 kinds of processes in a distributed program:
  - **Clients**
  - **Servers**
  - Filters
  - Peers

A client is a triggering process, a server is a reactive process. Clients make requests that trigger reactions from servers.

Different kind of processes

- Basically there are 4 kinds of processes in a distributed program:
  - Clients
  - Servers
  - **Filters**
  - Peers

A filter is a data transformer, receiving streams of data, transforms this data in some way, and sends the results to its output channels.
→ Kind of a server and client together

Different kind of processes

- Basically there are 4 kinds of processes in a distributed program:
  - Clients
  - Servers
  - Filters
  - **Peers**

A peer is one of a collection of identical process that interact to provide a service or solve a problem. For example, two peers might manage a copy of a replicated file and interact to keep both copies consistent, or they might cooperate to solve part of a parallel programming problem.

The challenge of distributed programs

- Each process has direct access only to its own data
- Thus, the challenge of distributed programs is:
- Maintaining or determining a **global state** of the distributed program

Async Msg Passing in JR (1/2)

JR book chapter 7

- Notion of *channels* is made through **operations**
- Processes **`send`** messages to and **`receive`** messages from **operations**
- **Operations** are therefore queues of messages
  - The operation has no corresponding method!
- Invocation of methods are serviced by **`receive`** statements
  - The process executing the **`receive`** statement delays if no invocation is present

FIFO

Async Msg Passing in JR (2/2)

JR book chapter 7

- A **send** invocation causes the invocation to be appended to the message queue
- The invoker continues immediately after the invocation has been sent (→ non–blocking)
- A **receive** statement specifies an operation and gives a list of zero or more variables separated by commas
  - `receive` *`op_expr(variable, variable,...)`*`;`
- The *`op_expr`* is any expression that evaluates to an operation
  - either the **operation name** or an **operation capability**

JR Example

As an example, consider a three process system. The program uses two operations, **stream1** and **stream2**. The first process sends its numbers (the numbers are ordered), including the end of stream marker EOS to **stream1**, the second to **stream2**.

The merge process first gets a number from each stream. It executes the body of the loop as long as one of the two numbers is not EOS. The if statement compares the two numbers, outputs the smaller, and receives the next number in the stream from which the smaller number came. If one stream ends before the other, v1 or v2 will be EOS. Since EOS is larger than any other number in the stream, numbers from the non-terminated stream will be consumed until it is also finished. The loop terminates when both streams have been entirely consumed.

```
public class StreamMerge {

  private static final int EOS =
    Integer.MAX_VALUE;// end of stream marker
  private static op void stream1(int);
  private static op void stream2(int);


  private static process one {
    int y = 38;
    send stream1(y);
    send stream1(56);
    send stream1(77);
    send stream1(83);
    send stream1(EOS);
  }


  private static process two {
    int y = 14;
    send stream2(y);
    send stream2(44);
    send stream2(98);
    send stream2(EOS);
  }

  private static process merge {
    int v1, v2;
    receive stream1(v1);
    receive stream2(v2);
    while (v1 < EOS || v2 < EOS) {
      if (v1 <= v2) {
        System.out.println(v1);
        receive stream1(v1);
      }
      else { // v1 > v2
        System.out.println(v2);
        receive stream2(v2);
      }
    }
    System.out.println(EOS);
  }

  public static void main(String [] args) {}

}
```

Message order

JR book chapter 7.1

- Messages sent from one process to another are **delivered in the order in which they are sent**
  - Attention: Non-deterministic process execution can affect message ordering
    - *See code example on next slide*

```java
import edu.ucdavis.jr.JR;

public class Order {

  private static op void a(int);
  private static op void b(int);


  private static process one {
    send a(1);
    send b(2);
  }

  private static process two {
    int x;
    receive a(x);
    send b(3);
  }

  private static process three {
    int x;
    receive b(x);
    System.out.println("three1 "+x);
    receive b(x);
    System.out.println("three2 "+x);
  }

  public static void main(String [] args) {
  }
}
```

```java
import edu.ucdavis.jr.JR;

public class Order {

  private static op void a(int);
  private static op void b(int);


  private static process one {
    send a(1);
    send b(2);
  }

  private static process two {
    int x;
    receive a(x);
    send b(3);
  }

  private static process three {
    int x;
    receive b(x);
    System.out.println("three1 "+x);
    receive b(x);
    System.out.println("three2 "+x);
  }

  public static void main(String [] args) {
  }
}
```

Output will be:

 # three1 2

 # three2 3

Invoking & Servicing via Caps

JR book chapter 7.2

- An operation serviced by a receive statement can be invoked indirectly via capabilities
- Some examples …

```
public class Cap1 {
  private static op void f(double);
  private static op void g(double);

  public static void main(String [] args) {
    cap void (double) y, z;

    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    // invoke what y points to and then what z points to
    send y(4.351);
    send z(8.21);
  }
  private static process pf {
    double d; receive f(d);
    System.out.println("pf got "+d);
  }
  private static process pg {
    double d; receive g(d);
    System.out.println("pg got "+d);
  }
}
```

- *Number of arguments affects the executed operation*
- *Caps used as sending operation*

Demo

```
public class Cap2 {
  private static op void f(double);
  private static op void g(double);

  public static void main(String [] args) {
    cap void (double) y, z;

    // make y point to one of f or g, and z point to other
    if (args.length > 0) { y = f; z = g; }
    else                 { y = g; z = f; }
    send f(4.351);
    send g(8.21);
    double d;
    receive y(d);
    System.out.println("rcv1 got "+d);
    receive z(d);
    System.out.println("rcv2 got "+d);
  }
}
```

- *Number of arguments affects the executed operation*
- *Caps used as receiving operation*

Demo

```
public class Cap3 {
  public static op void getcap(cap void (double));

  public static process p {
    op void f(double);
    send getcap(f);
    double d;
    receive f(d);
    System.out.println("rcv got "+d);
  }

  public static process q {
    cap void (double) y;
    receive getcap(y);
    send y(5.78);
  }

  public static void main(String [] args) {
  }
}
```

- *Operation `f` is local to process `p`. However, a capability for it is passed outside of `p` and used by process `q`.*
- *The capability is passed via the operation `getcap`, which is known to both `p` and `q`*

Semaphore – Take two

JR book chapter 7.5

- Semaphores (V() and P()) in JR are an abbreviation for **operations – send – receive statements**

Semaphore primitive	Corresponding message passing primitive
`sem s`	`op void s()`
`P(s)`	`receive s()`
`V(s)`	`send s()`

`sem s = `*`expr`*	`op void s();`
	`{`
	`    int v=expr;`
	`    for (int i=0; i < v; i++) send s();`
	`}`

```
public class CS {
  private static final int N = 20; // number of processes
  private static int x = 0;        // shared variable
  private static sem mutex = 1;    // mutual exclusion for x




  private static process p( (int i = 0; i < N; i++) ) {
    // non-critical section
    ...
    // critical section
    P(mutex); // enter critical section
    x = x + 1;
    V(mutex); // leave critical section
    // non-critical section
    ...
  }

  public static void main(String [] args) {
  }
}
```

```
public class CS {
  private static final int N = 20; // number of processes
  private static int x = 0;        // shared variable
  private static op void mutex();  // mutual exclusion for x

  static {
    send mutex(); // initialize mutex to "1"
  }

  private static process p( (int i = 0; i < N; i++) ) {
    // non-critical section
    ...
    // critical section
    receive mutex(); // enter critical section
    x = x + 1;
    send mutex();    // leave critical section
    // non-critical section
    ...
  }

  public static void main(String [] args) {
  }
}
```

Usage of P() and V() primitives makes the code somewhat more concise and readable

Shared Operations

JR book chapter 7.7

- Operations can be shared by multiple processes
- When operation is shared, processes compete for the operation’s invocation
- Processes obtain the invocation in a first-come – first-served basis

Shared Operations – Example

- Server work queues
- Share operation can be used to permit multiple server to service the same work queue

Example – adaptive quadrature for calculating the area

Given are a continuous, non-negative function f(x) and two values l and r, l < r. The problem is to compute the area bounded by f(x), the x axis, and the vertical lines through l and r. This corresponds to approximating the integral of f(x) from l to r. The following code uses a shared operation, called bag (bag of tasks); each task represents a sub-interval over which the integral of f(x) is to be approximated.

```
public class AQ {
  private static final int N = 20; // number of workers
  private static op void bag(double, double, double, double);
  private static op void result(double);
  private static final double Epsilon = 0.001; //conv test

  private static double f(double x) {
    return Math.pow(x,3.0);
  }

  private static double area = 0.0;
  private static process administrator {
    double part;
    double l = 0.0, r = 4.0;
    send bag(l,r,f(l),f(r));
    while (true) {
      receive result(part);
      area += part;
    }
  }

  private static process worker( (int i = 1; i <= N; i++) ) {
    double a, b, m, fofa, fofb, fofm;
    double larea, rarea, tarea, diff;
    while (true) {
      receive bag(a,b,fofa,fofb);
      m = (a+b)/2;  fofm = f(m);
        // compute larea, rarea, and tarea
        // using trapezoidal rule
        larea = (m - a) * (fofa + fofm) / 2.0;
        rarea = (b - m) * (fofm + fofb) / 2.0;
        tarea = (b - a) * (fofa + fofb) / 2.0;
        diff  = Math.abs(tarea - (larea + rarea));
        if (diff <= Epsilon) { /* diff small enough */
          send result(larea + rarea);
        }
        else { // diff > Epsilon /* diff too large */
          send bag(a, m, fofa, fofm);
          send bag(m, b, fofm, fofb);
        }
      }
    }

  public static void main(String [] args) {
    // register done as the quiescence operation
    try {
      JR.registerQuiescenceAction(done);
    } catch (QuiescenceRegistrationException e) {
      e.printStackTrace();
    }
  }
  private static op void done() {
    System.out.println("computed area = "+area);
  }
}
```

Elegant way to detect the end of all dynamically created processes

Demo

Client – Server Models

JR book chapter 7.3

- A disk server might read information from a disk
- Clients might pass it requests for disk, blocks and then wait for results.

Client – Server Models – Model 1

JR book chapter 7.3

```
import edu.ucdavis.jr.JR;
public class Model1 {
  private static op void request(char);
  private static op void results(double);

  private static process client {
    send request('w');
    // possibly perform some other work
    double d;
    receive results(d);
    System.out.println(" got " + d);
  }
```

```
  private static process server {
    while (true) {
      char data;  double ans;
      receive request(data);
      // handle request; put answer in ans
      ans = 222.8;
      send results(ans);
    }
  }

  public static void main(String [] args) {
  }
}
```

- Works for only one client and one server
- Multiple clients: one client can obtain the result intended for another
  - `result` operation would be shared by all clients

Client – Server Models – Model 2

JR book chapter 7.3

```
import edu.ucdavis.jr.JR;

public class Model2 {
  private static final int N = 20; // number of
client processes
  private static op void request(int, char);
  private static cap void (double) results[] =
    new cap void (double)[N];

  static {
    for (int i = 0; i < N ; i++ ) {
      results[i] = new op void (double);
    }
  }

  private static process client( (int i = 0;
                                  i < N; i++) ) {
    send request(i, 'w');
    // possibly perform some other work
    double d;
    receive results[i](d);
    System.out.println(i + " got " + d);
  }

  private static process server {
    while (true) {
      int id;  char data;  double ans;
      receive request(id, data);
      // handle request; put answer in ans
      ans = id * 2.0;
      send results[id](ans);
    }
  }

  public static void main(String [] args) {
  }
}
```

- Works for multiple clients
- Drawback, we need to know in advance how many clients we want to server (N=20)
- `results` array is a simple mean to associate an operation with each client process

Client – Server Models – Model 3

JR book chapter 7.3

```
import edu.ucdavis.jr.JR;
public class Model3 {
  private static final int N = 20; // number
of client processes
  private static op void request(cap void
(double), char);

  private static process client( (int i = 0;
                                  i < N; i++) ) {
    op void results(double);
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    System.out.println(i + " got " + d);
  }
```

```
  private static process server {
    while (true) {
      cap void (double) results_cap;
      char data;
      double ans;
      receive request(results_cap, data);
      // handle request; put answer in ans
      ans = data * 2.0; //not very interesting
      send results_cap(ans);
    }
  }

  public static void main(String [] args) {}
}
```

- Local operation for the client and passing the cap to `request` → `results_cap`
- Any number of clients, they just need to have a results op
- The server works in serial

Client – Server Models – Model 4

JR book chapter 7.3

```
import edu.ucdavis.jr.JR;
public class Model4 {
  private static final int N = 20; // number of client processes
  private static op void request(cap void (double), char);

  private static process client( (int i = 0;
                                  i < N; i++) ) {
    op void results(double);
    send request(results, 'w');
    // possibly perform some other work
    double d;
    receive results(d);
    System.out.println(i + " got " + d);
  }
```

```
  private static void request (
                 cap void (double) results_cap,
                 char data) {
    double ans;
    // handle request; put answer in ans
    ans = data * 2.0; //not very interesting
    send results_cap(ans);
  }

  public static void main(String [] args) {}
}
```

- The `request` operation plays the role of the server process
- `request` operation is now serviced by a method
- Multiple servers → new server is created for each client's request

Parameter Passing Details JR

JR book chapter 7.8

- Parameter passing in JR op invocation on the same VM follow the same rules as in pure Java (method invocation):

→ Parameters are passed **by value**

- Even **object references** are passed by value

```
public class BasicArraySend {
  public static void main(String [] args) {
    op void f(int []);

    // generate two invocations of f
    int [] b = new int [2];
    b[0] = 11; b[1] = 34;
    send f(b);
    b[0] = 65; b[1] = 87;
    send f(b);
    // service two invocations of f
    for (int k = 1; k <= 2; k++) {
      inni void f(int [] a) {
        for (int i = 0; i < 2; i++) {
          System.out.println(k +" a["+i+"] "+a[i]);
        }
      }
    }
  }
}
```

Output will be:

```
# 1 a[0] 65
# 1 a[1] 87
# 2 a[0] 65
# 2 a[1] 87
```