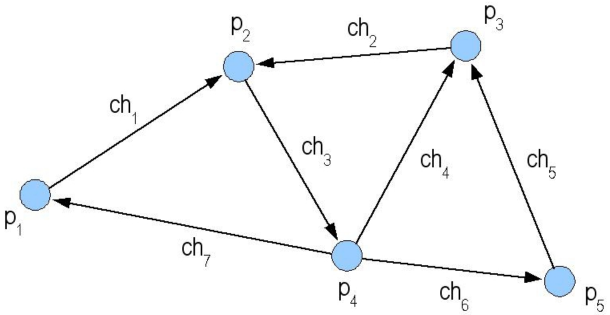
Model of distributed computing

static system



G = (V,E)

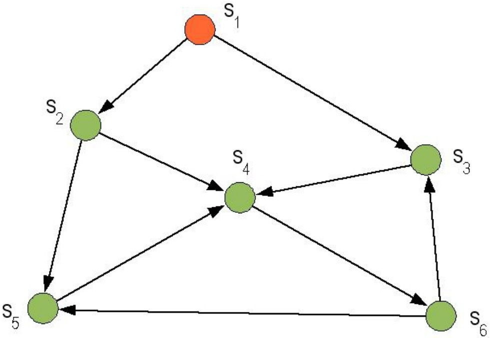
G - oriented graph

$V = \{ p_1, p_2, \ldots \}$ - processes

$E = \{ ch_1, ch_2, \ldots \}$ - communication among processes

Process model

finite state description



FSM = (S, s_1, T)

$S = \{ s_1, s_2, \ldots \}$

s_1

$T = \{ t_1, t_2, \ldots \}$

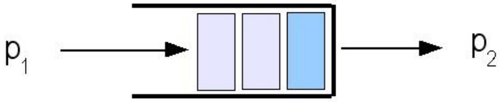
FSM - Finite State Machine

- states
- starting state
- transitions

Communication channel model

behavior

- poit-to-point connection
- FIFO strategy



parallel computation

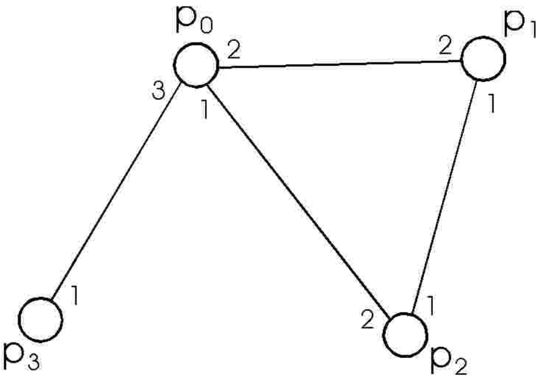
synchronnous channel = blocking sender

distributed computation

asynchronnous channel = nonblocking sender

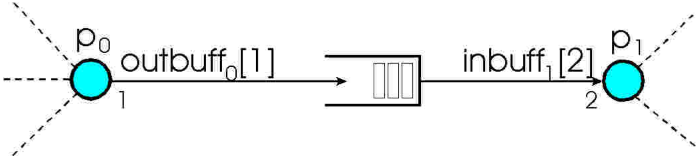
Model of execution

description



Model of execution

for asynchronnous communication



Model of execution

for asynchronnous communication

processes

$P = \{p_0, , , p_{N-1}\}$

state set of the processes

$Q_i = \{q_{i_0}, , , q_{i_{M_i}}\}$

communication actions

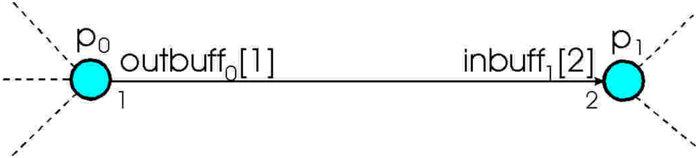
$CA_i = \{inbuff_i[1], , , inbuff_i[r], outbuff_i[1], , , outbuff_i[r]\}$

state set of communication actions

. . . . .

Model of execution

for synchronnous communication



Model of execution

for synchronnous communication

processes

$P = \{p_0, , , p_{N-1}\}$

state set of the process

$Q_i = \{q_{i_0}, , , q_{i_{M_i}}\}$

communication actions

$CA_{i,j} = \{outbuff_i[k]\text{->}inbuff_j[l], ...\}$

state set of communication actions

. . . . .

Model of execution

description

computation actions

$C_k = comp(i)$

communication actions

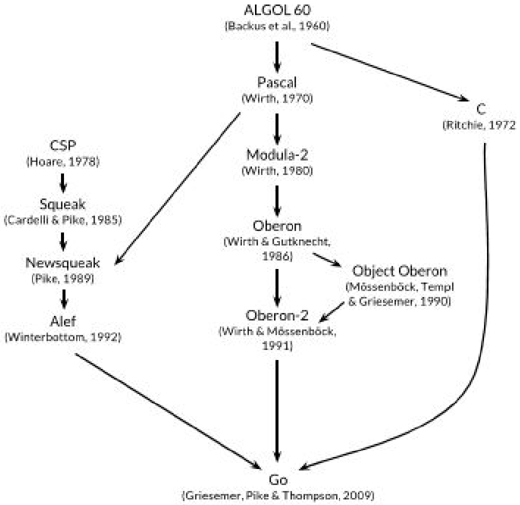
$\phi_k = out(i, m)$ - asynchronnous action
$\phi_k = in(j, m)$

$\phi_k = del(i, j, m)$ - synchronnous action

execution

$C_0, \phi_1, C_1, \phi_2, C_2, \phi_3, \ldots$

Go language


ALGOL 60
(Backus et al., 1960)
Pascal
(Wirth, 1970)
C
(Ritchie, 1972
CSP
(Hoare, 1978)
Modula-2
(Wirth, 1980)
Squeak
(Cardelli & Pike, 1985)
Oberon
(Wirth & Gutknecht, 1986)
Newsqueak
(Pike, 1989)
Object Oberon
(Mössenböck, Templ & Griesemer, 1990)
Alef
(Winterbottom, 1992)
Oberon-2
(Wirth & Mössenböck, 1991)
Go
(Griesemer, Pike & Thompson, 2009)

Go language

distributed computation support

gocoroutines

fname () - calling routine

go fname() - calling routine without waiting

Go language
distributed computation support

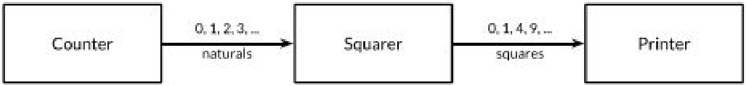
channels

ch := make(chan int) - creating the unbuffered channel
ch := make(chan int, 0) - creating the unbuffered channel
ch := make(chan int, 3) - creating the buffered channel

ch <- x - sending the statement
y <- ch - receiving a message
<- ch - receiving a message and discarding it

close(ch) closing the channel

Go language
example



Go language

example

```
func counter(out chan<- int) {
    for x:=0;x<100;x++ {
        out <- x
    }
    close(out)
}
func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}
```

Go language

example

```
func printer(in <-chan int) {
    for v := range in {
        fmt.println(v)
    }
}
```

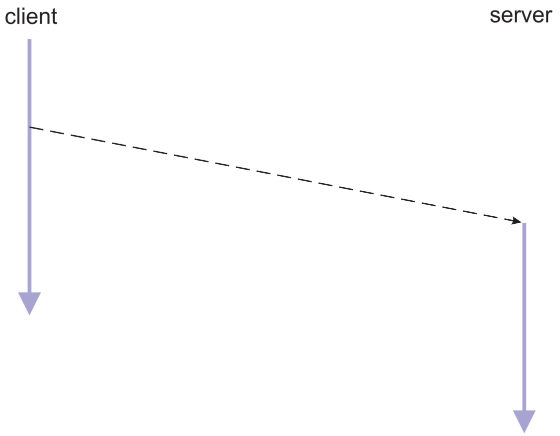
Go language

example

```
func main() {
    naturals := make(chan int)
    squares := make(chan int)
    go counter (naturals)
    go squarer (squares, naturals)
    printer (squares)
}
```

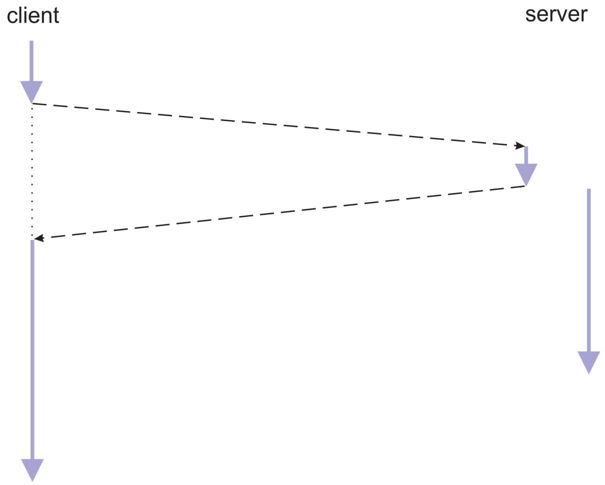
Java RMI programming support for asynchrony

Fire and Forget

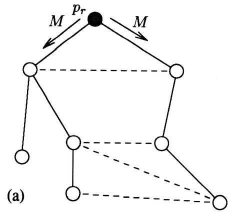


Java RMI programming support for asynchrony

Sync with Server



Broadcast

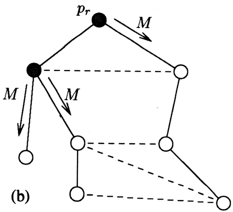


p_r

no message received:

send M to all children

terminate



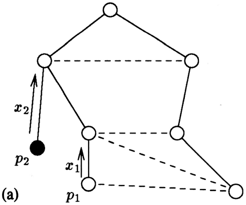
$p_i, 0 \leq i \leq n-1, i \neq r$:

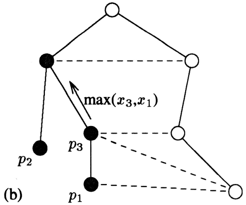
message M received from the parent:

send M to all children

terminate

Convergecast



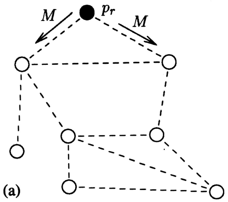


p_i having no child
- no message received:
  - send x_i to the parent
  - terminate

p_i having children:
- message x_j received from the child:
  - **if** messages x_j received from all children
    - send max(x_j) to the parent
    - terminate

p_r:
- message x_j received from the child:
  - **if** messages received from all children
    - evaluate max(x_j)
    - terminate

Flooding

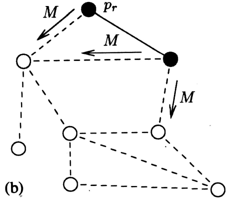


p_r

no message received:

send M to all neighbours

terminate



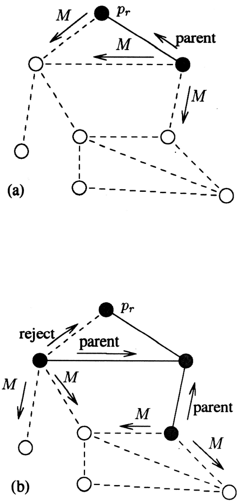
$p_i, 0 \leq i \leq n-1, i \neq r$:

message M received from any neighbour j:

send M to all neighbours excluding j

terminate

Construction of Spanning Tree (1)



no message received:

  if $i = r$ and $parent = nil$

    send M to all neighbors

    $parent := i$

receiving M from p_j:

  add j to *others*

  if $parent = nil$

    $parent := j$

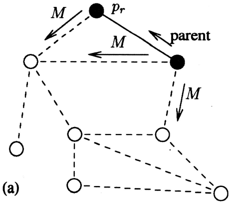
    send *parent* to p_j

    send M to all neighbors except p_j

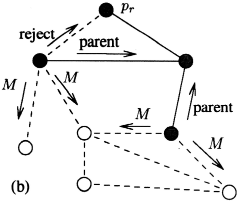
  else

    send *reject* to p_j

Construction of Spanning Tree (2)



receiving *parent* from p_j:
add j to *children*
if *children*&*others* contains all neighbors
except *parent*
terminate



receiving *reject* from p_j:
add j to *others*
if *children*&*others* contains all neighbors
except *parent*
terminate

Depth-First Search Spanning Tree with a Specified Root (1)

no message received:

    if $i = r$ and $parent = nil$

        $parent := i$

        select p_j from *unexplored*

            remove p_j from *unexplored*

            send M to p_j

Depth-First Search Spanning Tree with a Specified Root (2)

receiving M from p_j:
    if *parent* = *nil*
        parent := j
        remove p_j from *unexplored*
        if *unexplored* $\neq 0$
            select p_k from *unexplored*
                remove p_k from *unexplored*
                send M to p_k
        else send *parent* to *parent*
    else send *reject* to p_j

Depth-First Search Spanning Tree with a Specified Root (3)

receiving *parent* or *reject* from p_j:

    if received *parent*

        add p_j to *children*

    if *unexplored* = 0

        if $parent \neq i$

            send *parent* to *parent*

            terminate

    else

        select p_k from *unexplored*

            remove p_k from *unexplored*

            send M to p_k

Depth-First Search Spanning Tree without a Specified Root (1)

no message received:

    if *parent* = *nil*

        leader := *id*

        parent := *i*

        select p_j from *unexplored*

            remove p_j from *unexplored*

            send *leader* to p_j

Depth-First Search Spanning Tree without a Specified Root (2)

receiving *new-id* from p_j:

 if $\mathit{leader} < \mathit{new\text{-}id}$

  $\mathit{leader} := \mathit{new\text{-}id}$

  $\mathit{parent} := j$

  $\mathit{unexplored}$:= all neighbors of p_i except p_j

  if $\mathit{unexplored} \neq 0$

   select p_k from *unexplored*

    remove p_k from *unexplored*

    send *leader* to p_k

  else send *parent* to *parent*

 else

  if $\mathit{leader} = \mathit{new\text{-}id}$

   send *already* to p_j

Depth-First Search Spanning Tree without a Specified Root (3)

```
receiving parent or already from p_j:
    if received parent
        add p_j to children
    if unexplored = 0
        if parent ≠ i
            send parent to parent
        else
            terminate
    else
        let p_j from unexplored
            remove p_j from unexplored
            send leader to p_j
```