149th NII Shonan Meeting

Distributed Algorithms for Robot Systems

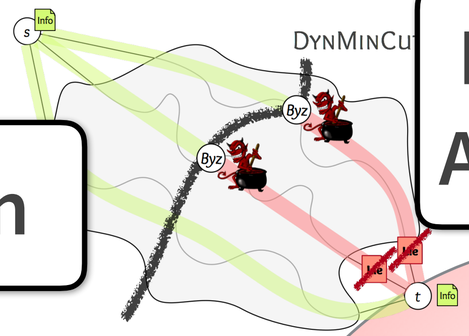
Xavier Défago

School of Computing

Tokyo Institute of Technology

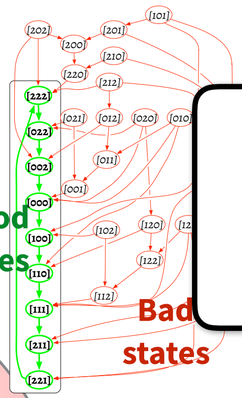
May 2019

Research Area(s)

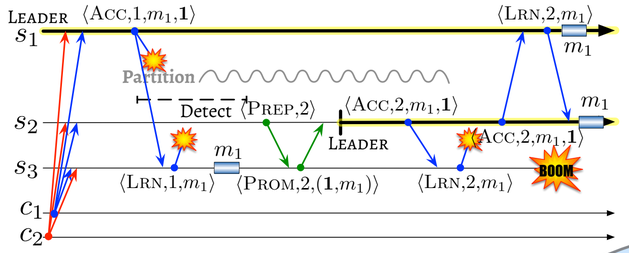


Blockchain

Byzantine Agreement



Gathering, Pattern Formation

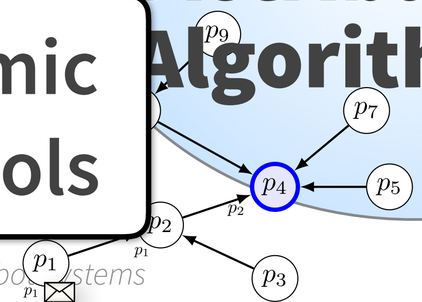


Dependable Systems

Self-Stabilization

```
class Arbitr
{
  private de
  private va
  private va
  private va
  private va
  private var announced = false

  private def visitNextChild() {
    children match {
      case next :: tail    => SEND (Token(me, next,   maxID)) ; children = tail  // Rule 1
      case Nil if ! isRoot => SEND (Token(me, parent, maxID))                    // Rule 2
      case Nil if isRoot   => SEND (AnnounceLeader(me, neighbors+me, me))        // End
  } }

  def onSend = {
    case Candidate if color == Red   => visitNextChild()
    case Candidate if color == Black => /* IGNORE */
  }

  listenTo(classOf[Token])
  listenTo(classOf[AnnounceLeader])
  def onReceive = {
    case Token(_,_,pid,_) if pid < ma
    case Token(_,_,pid,_) if pid == m
    case Token(from,_,pid,_)
      color = Black ; maxID = pid ; pa
      children = (neighbors - from).t
      visitNextChild()

    case AnnounceLeader(_,_,_,_) if a
    case AnnounceLeader(from,_,leader
      announced = true
      SEND (AnnounceLeader(me, neighb
      DELIVER (Elected(Some(leader)))
} }
```

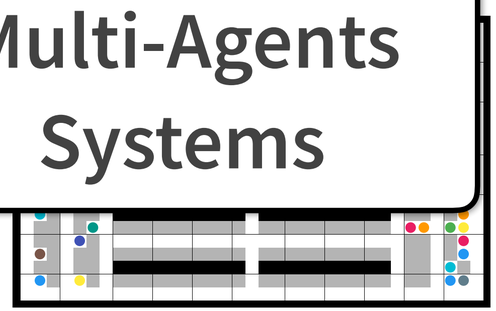
Distributed Algorithms



Epidemic Protocols

Multi-Robots Systems

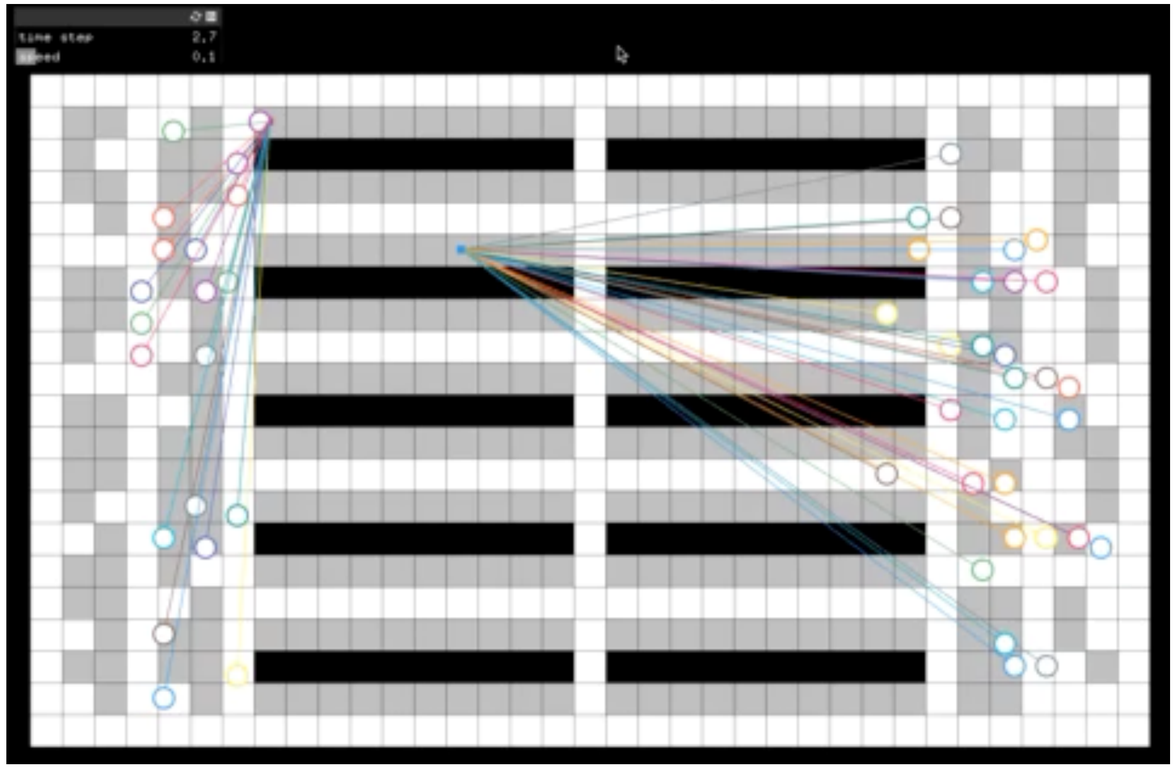
Cooperati Robotics



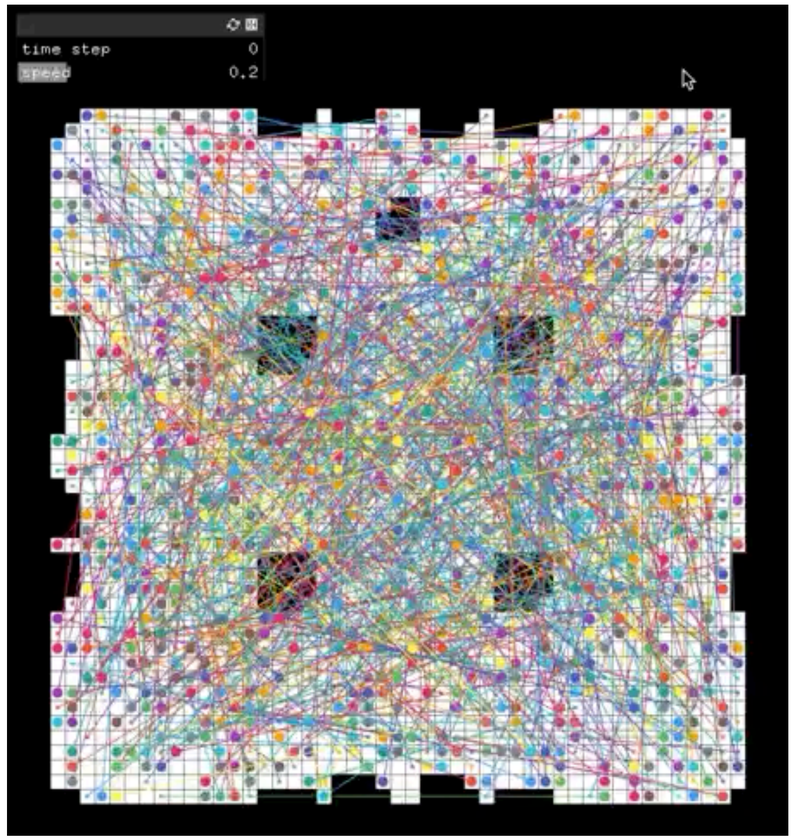
Multi-Agents Systems

Multi-Agent Systems

Pickup and Delivery



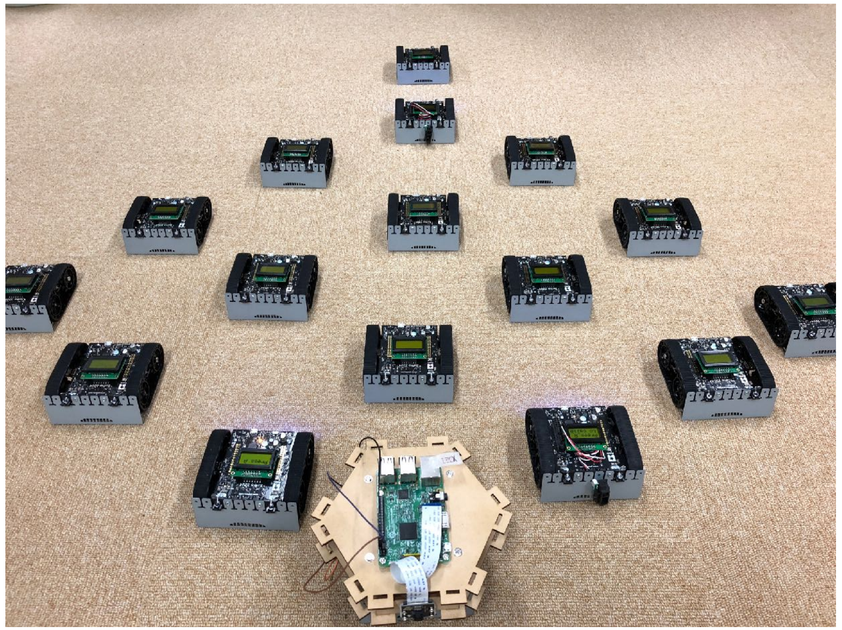
Path Finding

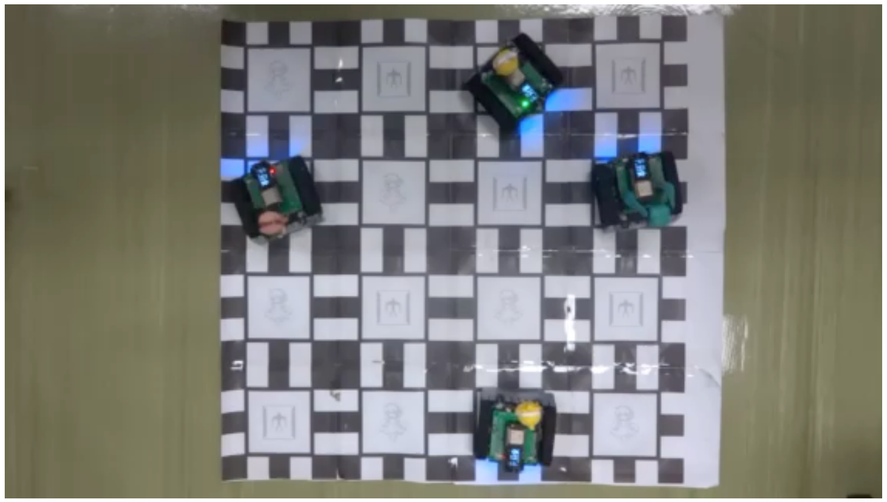


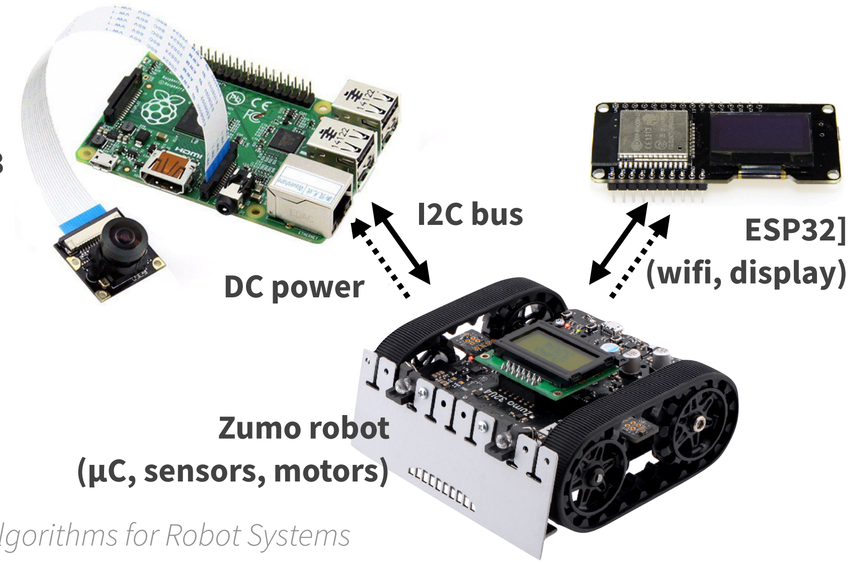
K.Okumura, M.Machida, X.Défago, Y.Tamura: **Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding**. IJCAI 2019, to appear

Prototyping Platform

- **Robot Platform**
  - 17 robots
  - Zumo + ESP32 / Raspberry Pi 3
  - used for prototyping
  - fully decentralized control
  - local communication





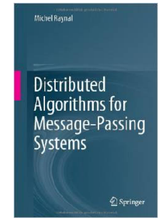


Non-Research

Neko > ScalaNeko > Ocelot

- **What?**
  - DSL for Distributed Algorithms
- **Purpose**
  - **Teaching** / Prototyping
- **Timeline**
  - 2000's Neko framework (EPFL/JAIST) w/Peter Urban
  - 2008 Scala wrappers for Neko (JAIST)
  - 2012 ScalaNeko (JAIST)
  - 2017 Ocelot (Tokyo Tech)

Models



Active *(spanning tree)*

when $r = 1, 2, \ldots, D$ **do**
begin synchronous round
(1) **for each** $j \in neighbors_i$ **do** send UPDATE$(length_i)$ to p_j **end for**
(2) **for each** $j \in neighbors_i$ **do** receive UPDATE$(length_j)$ from p_j **end for**
(3) **for each** $k \in \{1 \ldots n\} \setminus \{i\}$ **do**
(4) **let** $length\_ik = \min_{j \in neighbors_i}(lg_i[j] + length_j[k])$
(5) **if** $(length\_ik < length_i[k])$ **then**
(6) $length_i[k] \leftarrow length\_ik$
(7) $routing\_to_i[k] \leftarrow$ a neighbor that realizes the previous minimum
(8) **end if**
(9) **end for.**
end synchronous round

Blocking *(random leader elect.)*

1 $rd_i \leftarrow$ random$(1, n)$;
2 broadcast RANDOM(rd_i);
3 **wait for** a message RANDOM(rd_x) **from each** process p_x;
4 $elected_i \leftarrow (\sum_{x=1}^{n} rd_x) \bmod (n + 1)$.

Reactive *(mutex: drinking philosophers)* Models

```
1  operation acquire_mutex() is
2      cs_state_i ← trying;
3      for each j ∈ R_i do send REQUEST() to p_j;
4      wait (R_i = ∅);
5      cs_state_i ← in;
6      for each j ∈ {1, ..., n} \ {i} do perm_state_i[j] ← used; end for.
7
8  operation release_mutex() is
9      cs_state_i ← out;
10     for each j ∈ perm_delayed_i do send PERMISSION({i, j}) end for;
11     R_i ← perm_delayed_i;
12     perm_delayed_i ← ∅.
13
14 when REQUEST() is received from p_j do
15     prio_i ← (cs_state_i = in)
16                 ∨((cs_state_i = trying) ∧ (perm_state_i[j] = new))
17                 ∨((cs_state_i = trying) ∧ (j ∈ R_i));
18     if (prio_i) then
19         perm_delayed_i ← perm_delayed_i ∪ {j}
20     else
21         send PERMISSION({i, j}) to p_j
22         R_i ← R_i ∪ {j};
23         if (cs_state_i = trying) then send REQUEST() to p_j end if
24     end if.
25
26 when PERMISSION({i, j}) is received from p_j do
27     R_i ← R_i \ {j}
28     perm_state_i[j] ← new.
```

Active *(spanning tree)*

```
when r = 1, 2, ..., D do
begin synchronous round
(1)  for each j ∈ neighbors_i do send UPDATE(length_i) to p_j end for
(2)  for each j ∈ neighbors_i do receive UPDATE(length_j) from p_j end for
(3)  for each k ∈ {1...n} \ {i} do
(4)    let length_ik = min_{j∈neighbors_i}(lg_i[j] + length_j[k])
(5)    if (length_ik < length_i[k]) then
(6)      length_i[k] ← length_ik
(7)      routing_to_i[k] ← a neighbor that realizes the previous minimum
(8)    end if
(9)  end for.
end synchronous round
```

Blocking *(random leader elect.)*

```
1  rd_i ← random(1, n);
2  broadcast RANDOM(rd_i);
3  wait for a message RANDOM(rd_x) from each process p_x;
4  elected_i ← (∑_{x=1}^{n} rd_x) mod (n + 1).
```

$$\textbf{wait } (R_i = \emptyset);$$

HelloWorld

```
class GreeterA(p: ProcessConfig) extends ActiveProtocol(p)
{
  listenTo(classOf[Greetings])
  def run() {
    SEND(Greetings(me, neighbors, "Hello Shonan!"))
    for (_ <- neighbors) {
      Receive {
        case Greetings(from, _, msg) => println(s"$me got from $from: $msg")
      }
    }
  }
}

case class Greetings(from: PID, to: Set[PID], msg: String)
  extends MulticastMessage

object HelloWorldMain
  extends Main(topology.Grid(2,2))(ProcessInitializer { new GreeterA(_) })
```

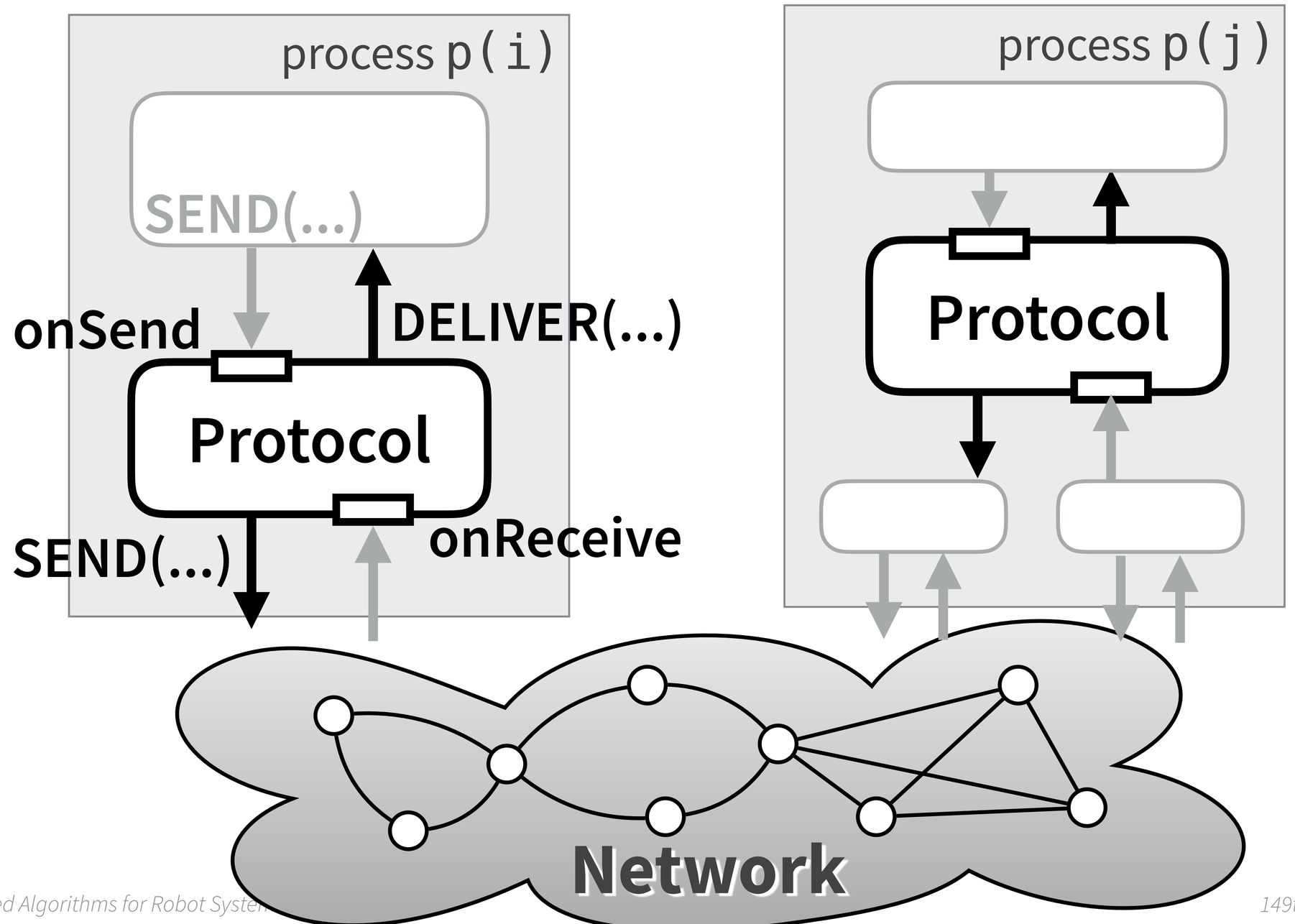
HelloWorld (react)

```
class GreeterR(p: ProcessConfig) extends ReactiveProtocol(p)
{
  override def preStart() = {
    SEND(Greetings(me, neighbors, "Hello Shonan!"))
  }

  listenTo(classOf[Greetings])
  def onReceive = {
    case Greetings(from, _, msg) => println(s"$me got from $from: $msg")
  }
  def onSend = PartialFunction.empty
}

case class Greetings(from: PID, to: Set[PID], msg: String)
  extends MulticastMessage

object HelloWorldMain
  extends Main(topology.Grid(2,2))(ProcessInitializer { new GreeterR(_)})
```

Hello World (hybrid)

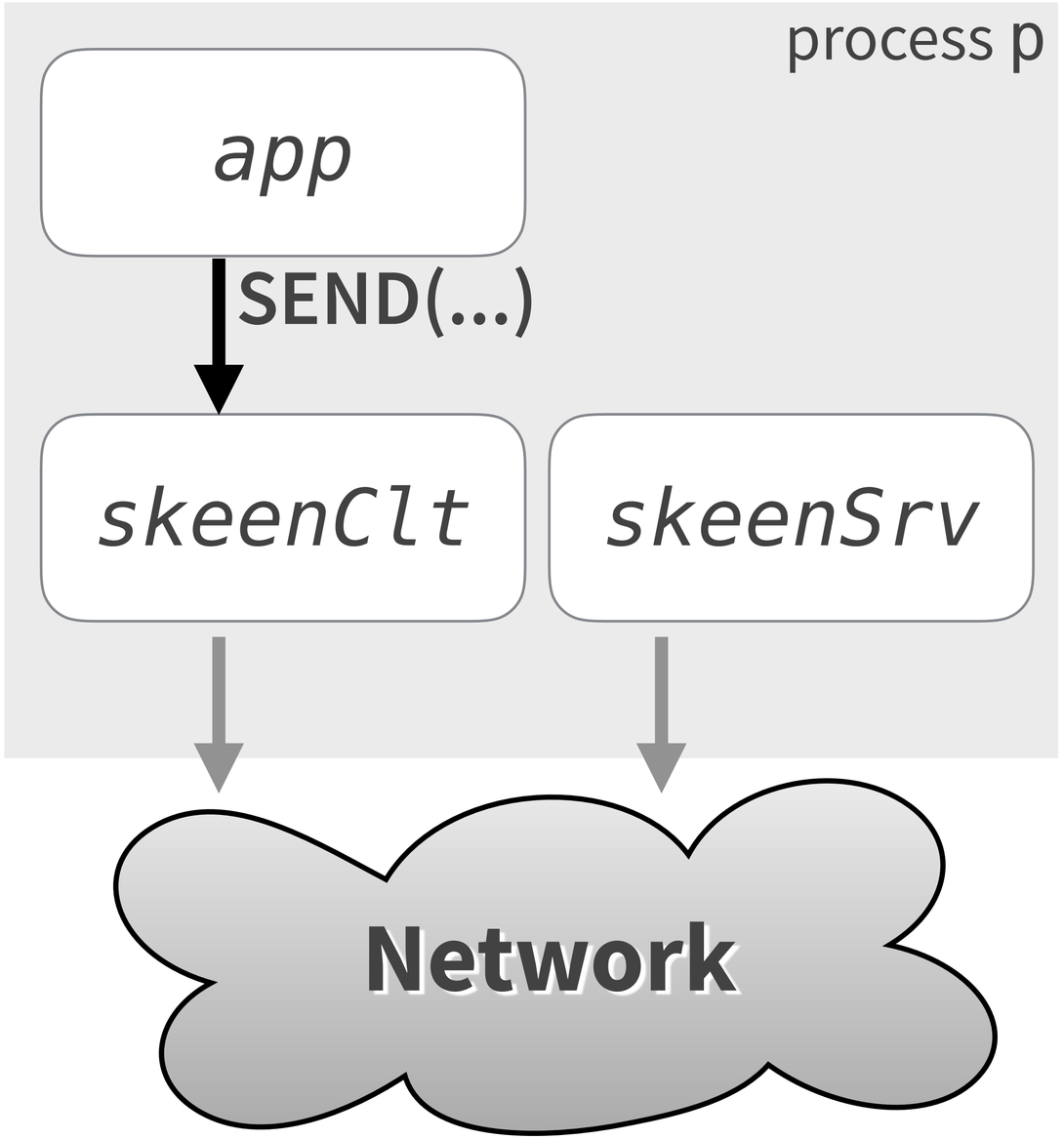
```
class GreeterH(p: ProcessConfig) extends ActiveProtocol(p)
{
  def run() {
    SEND(Greetings(me, neighbors, "Hello World!"))
  }

  listenTo(classOf[Greetings])
  override def onReceive = {
    case Greetings(from, _, msg) => println(s"$me got from $from: $msg")
  }
}

case class Greetings(from: PID, to: Set[PID], msg: String)
  extends MulticastMessage

object HelloWorldMain
  extends Main(topology.Grid(2,2))(ProcessInitializer { new GreeterH(_) })
```

Overview

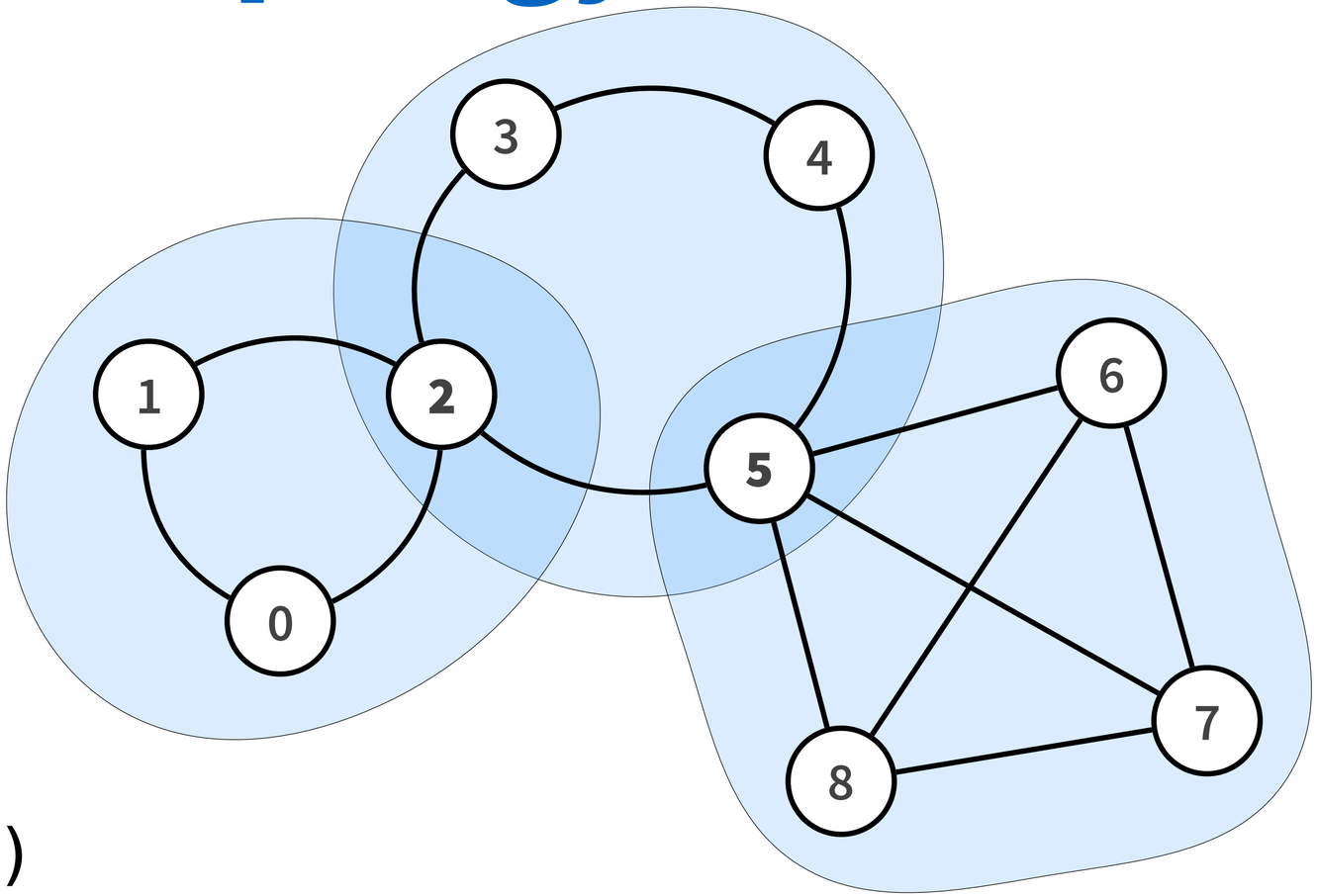

process p(i)
SEND(...)
onSend
DELIVER(...)
Protocol
SEND(...)
onReceive
process p(j)
Protocol
Network

Initializer & Composition

```
object SkeenTestApp
  extends Main(topology.Clique(10))(
  ProcessInitializer { p =>
    val app       = new BroadcasterApp(p)
    val skeenClt  = new Skeen.Client(p)
    val skeenSrv  = new Skeen.Server(p)

    app --> skeenClt
  })
```



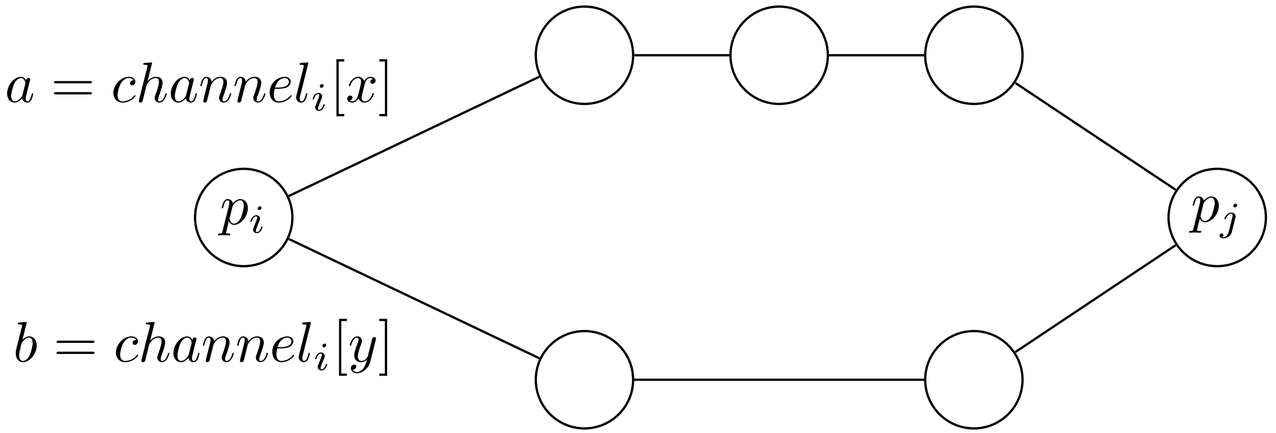
System Topology




```
object CutVerticesMain
  extends Main(
    topology.Clique(0 to 2)
      union topology.Ring(2 to 5)
      union topology.Clique(5 to 8)
  )(ProcessInitializer { new CutVertices(_) })
```

Cut Vertices

```
private var r_i = 0
private var inf_i = Set(me)
private var new_i = Set(me)
private var com_with_i = neighbors
private var routing_to_i = Map(me -> me)
private var dist_i       = Map(me -> 0)
private var relationR =
  neighbors.zipWithIndex.map(pi => pi._1 -> pi._2).toMap

def run() {
  while( new_i.nonEmpty ) {
    /* begin asynchronous round */
    r_i += 1
    SEND(Msg(me, com_with_i, r_i, new_i))
    new_i = Set.empty

    for (x <- com_with_i) {
      ReceiveOnly {
        case Msg(`x`,_,r,new_m) if r == r_i =>
          if (new_m.isEmpty) { com_with_i -= x }
          else {
            val aux = new_m diff (inf_i union new_i)
            for (id <- new_m) {
              if (! (inf_i union new_i).contains(id)) {
                routing_to_i = routing_to_i.updated(id, x)
                dist_i       = dist_i.updated(id, r_i)
              }
              else if (dist_i.get(id).exists(d => r_i == d || r_i == d+1)) {
                val y = routing_to_i(id)
                /* x,y in same biconnected component */
                relationR = transitiveAppend(relationR, (x,y))
            } }
            new_i = new_i union aux
    } }   }
    inf_i = inf_i union new_i
    /* end asynchronous round */
  }
  SEND ( Msg(me, com_with_i, r_i+1, new_i) )
}
```



Reactive *(mutex: drinking philo)* Models

```
1  operation acquire_mutex() is
2      cs_state_i ← trying;
3      for each j ∈ R_i do send REQUEST() to p_j;
4      wait (R_i = ∅);
5      cs_state_i ← in;
6      for each j ∈ {1, ..., n} \ {i} do perm_state_i[j] ← used; end for.
7
8  operation release_mutex() is
9      cs_state_i ← out;
10     for each j ∈ perm_delayed_i do send PERMISSION({i, j}) end for;
11     R_i ← perm_delayed_i;
12     perm_delayed_i ← ∅.
13
14 when REQUEST() is received from p_j do
15     prio_i ← (cs_state_i = in)
16                 ∨((cs_state_i = trying) ∧ (perm_state_i[j] = new))
17                 ∨((cs_state_i = trying) ∧ (j ∈ R_i));
18     if (prio_i) then
19         perm_delayed_i ← perm_delayed_i ∪ {j}
20     else
21         send PERMISSION({i, j}) to p_j
22         R_i ← R_i ∪ {j};
23         if (cs_state_i = trying) then send REQUEST() to p_j end if
24     end if.
25
26 when PERMISSION({i, j}) is received from p_j do
27     R_i ← R_i \ {j}
28     perm_state_i[j] ← new.
```

$$\textbf{wait } (R_i = \emptyset);$$

Drinking Philosophers

```
class ChandyMisra(p: ProcessConfig)
  extends ReactiveProtocol(p)
{
  private var cs_state_i: ChannelState = OUT
  private var perm_delayed_i = Set.empty[PID]
  private var perm_state_i = ALL.map(_ -> USED).toMap
  private var R_i = ALL.filter(me < _)

  def onSend = {
    case MutexClient.Request =>
      cs_state_i = TRYING
      SEND( Request(me, R_i) )
      if (R_i.isEmpty) enterCS()

    case MutexClient.Release =>
      cs_state_i = OUT
      perm_delayed_i.foreach(j => SEND(Permission(me, j)))
      R_i = perm_delayed_i
      perm_delayed_i = Set.empty
  }

  private def enterCS() {
    assert(R_i.isEmpty)
    cs_state_i = IN
    perm_state_i = perm_state_i.mapValues(_ => USED)
    DELIVER(MutexClient.CanEnter)
  }
```

```
  listenTo(classOf[Request])
  listenTo(classOf[Permission])
  def onReceive = {
    case Request(p_j, _) =>
      val prio_i =
        (cs_state_i == IN) ||
          ((cs_state_i == TRYING) &&
              ((perm_state_i(p_j) == NEW) ||
               (R_i contains p_j)))
      if (prio_i) { perm_delayed_i += p_j }
      else {
        SEND( Permission(me, p_j) )
        R_i += p_j
        if (cs_state_i == TRYING)
          SEND( Request(me, Set(p_j)))
      }

    case Permission(j, i) =>
      R_i -= j
      perm_state_i = perm_state_i.updated(j, NEW)
      if (R_i.isEmpty) enterCS()

    case _ => /* ignore */
  }
}
```

Thank You

- **Requirements**
  - Support many models
  - Support diff. levels of abstraction
  - Reduced boilerplate
  - Readable
  - Flexibility / composable
  - Robustness
  - Must run on laptop