Looking for a definition for "Dynamic Distributed System" (Yet another holy grail quest?)

Michel Raynal

IRISA, Université de Rennes, France

raynal@irisa.fr

My view of distributed computing (1)

- Real-time: masters On-time computing
- Parallelism: provides Efficiency
- Distributed computing:

$$\boxed{\text{masters} \quad \text{Uncertainty}}$$

(We are -more or less- implicitly using a lot of heuristics!)

My view of distributed computing (2)

Uncertainty is created by:

- Multiple loci of control
- Asynchrony (vs Synchrony)
- Failures (Failure models)
- Locality
- Process mobility (and related stuff)
- Low computing capacity, bandwidth
- Dynamicity (and Self-*, and then *-*!)
- Etc., etc., ..., etc.!

Personal view

- *Collapse: How Societies Choose to Fail or Survive*

  by Jared Diamond, 593 pages, Viking Press, 2005
  ISBN 0143036556

Static systems (1)

- Fixed number of processes
- Non-anonymous processes
- Communication model: Shared memory/message-passing
- Time model: synchronous/asynchronous
- Failure model: failure free/crash/omission/Byzantine
- Process initial knowledge

Static systems (2)

- Lots of constructive results (algorithms)
- Lots of impossibility results (wrt failure modes)
  - ⋆ Failure-free: leader election in anonymous systems
  - ⋆ Crash: consensus, k-set agreement, et cetera
  - ⋆ Omission, Byzantine
- Lower bounds, and lots of "best" solutions
- Nice tutorials, nice textbooks

Dynamic systems (1)

- P2P
  - ⋆ Originated from file sharing applications
  - ⋆ Resource discovery
  - ⋆ Best effort semantics
  - ⋆ Towards full-fledged computing platforms?

  -Liben-Nowell D., Balakrishnan H., and Karger D.R., Analysis of the Evolution of Peer-to-peer Systems. *Proc. 21th ACM PODC*, pp 233-242, 2002

- Grid computing, Clouds computing,
- Oportunistic computing, et cetera
- Lots of experimental results, but very few applications with proved deterministic properties

Please, have a look at
"The future of computing: logic or biology" by Leslie Lamport, 2003

Dynamic systems (2)

- Processes can enter and leave the system at will
- Enter/leave are (partially) uncontrolled
- Initial process knowledge is partial
- Locality is a basic principle
- Some applications (any non-trivial application?) require(s) some form of *stability* (that has to occur "often enough" in order to be able to do something)

Dynamic system vs large scale system

- Large scale systems (e.g., some types of sensor networks) are not necessarily dynamic
- "Large scale" and "Dynamicity" seem to be two distinct concepts (that are often confused)

What is “the most” important

The most fundamental concept seems to be

$\boxed{\text{Ability to adapt to the environment}}$

- Large scale: adaptability wrt number of participants

  (well-known pb in classical computing: design an efficient n-mutex algorithm from a 2-mutex algorithm)

- Dynamicity: adaptability wrt enter/leave (time)
- Fault-tolerance: adaptability wrt failures
- Self-organization: adaptability wrt structure
- Et cetera

Part I

From a static to a dynamic system

Aims

- Understand the "nature" of Dynamicity
- Provide methodology elements to help "going" from algorithms designed for static systems to algorithms for dynamic systems
- Illustrate it with examples

Static system model

- n processes, at most $t < n$ may crash
- Each process has an id, all ids are known by each process
- Communication graph: fully connected
- Communication channels are (more or less) reliable
- No upper bound on computation time
- No upper bound on message transfer delay

Algorithms in the static model

- A process can benefit from the knowledge of n and t
- $(n - t)$ is a key value when designing an algorithm: a process can wait for messages from $n-t$ processes while being sure it will not be blocked forever
- The parameters n and t define a liveness guarantee

Two communication abstractions in the static model

- Query/Response
  - ⋆ Broadcast a query
  - ⋆ Wait for the first $n-t$ responses (winning responses)
- Reliable broadcast

  -Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press, pp. 97-145, 1993

  - ⋆ Broadcast/Deliver
  - ⋆ If a message is delivered by a process, it is delivered by all correct processes

A type of dynamic system

- Same asynchrony assumptions as for the static model
- Finite arrival model:

  The system has infinitely many processes, but each run has only finitely many

  - Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, LNCS #1914, pp. 164-178, 2000

  - Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004

- Processes may join and leave the system at any time
- The system parameters n and t no longer exist

Facing dynamicity

- Stable instead of correct
  - ⋆ Replace the notion of correct process with the notion of stable process: a process that, once it has entered the system, neither crashes nor leaves
  - ⋆ Let $STABLE$ be the set of stable processes
- New progress condition
  - ⋆ Replace $(n - t)$ by the system parameter α
  - ⋆ Progress is guaranteed as long as $STABLE \geq \alpha$

Eventual leadership property

- **Static system**: There is a finite time τ and a correct process p such that, after τ, all the invocations of $leader()$ by any process returns always p
- **Dynamic system**: There is a finite time τ and a stable process p such that, after τ, all the invocations of $leader()$ by any process returns always p

On the system assumption side

- Static system: There is a time τ, a correct process p, and a set Q such that $\forall\ \tau' \geq \tau$:
  - $\star$ $|Q| = t + 1$, and
  - $\star$ Each time a process $\in Q$ issues a Query/Response, it receives a winning response from p
- Dynamic system: There is a time τ, a stable process p, and a set Q such that $\forall\ \tau' \geq \tau$:
  - $\star$ $Q \subseteq up(\tau')$, and
  - $\star$ Each time a process $\in Q$ issues a Query/Response, it receives a winning response from p
  - $\star$ Each time a process $\notin Q$ issues a Query/Response, it receives a winning response from a process $\in Q$

Algorithms

- A Time-free Assumption to Implement Eventual Leadership. *Parallel Processing letters*, 16(2):189-208, 2006 (Mostefaoui, Mourgaya, Raynal and Travers.)
- From Static Distributed Systems to Dynamic Systems. *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, IEEE Computer Society Press, pp. 109-119, 2005 (Mostefaoui, Raynal, Travers, Peterson, El Abbadi and Agrawal)
- The second paper uses the previous methodology to extend the first one to dynamic systems

Atomic objects in a dynamic system

- Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message-Passing Systems. *JACM*, 42(1):129-142, 1995

- Lynch, N. and Shvartsman A., RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 173-190, 2002.

- Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004

- Friedman R., Raynal M., Travers C., Abstractions for Implementing Atomic Objects in Distributed Systems. *9th Int'l Conference on Principles of Distributed Systems (OPODIS'05)*, Springer Verlag LNCS #3974:73-87, 2005

Summary

- Computation model
  - ⋆ Infinite nb of clients
  - ⋆ Atomic objects and infinite nb of servers
- Dynamic Read/Write quorums
- Persistent reliable broadcast
- Implementing read/write operations
- Practical instantiations

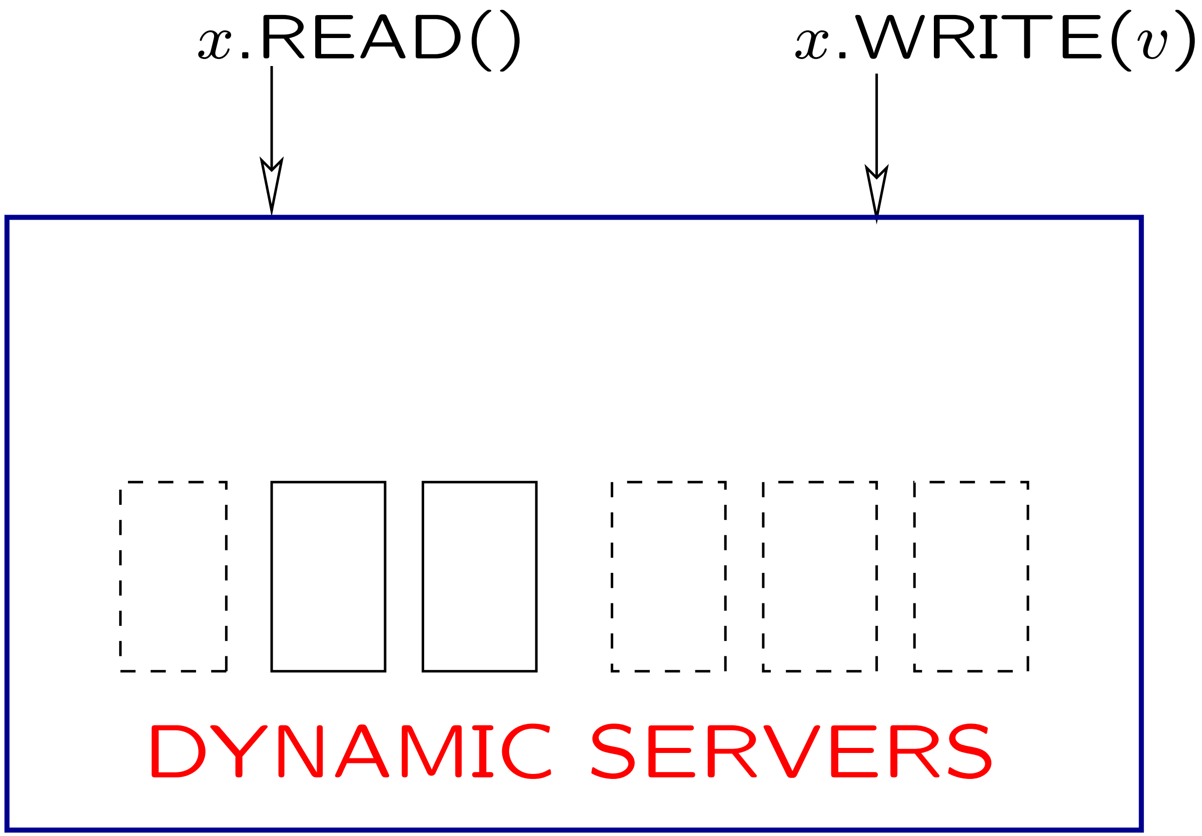
Dynamic systems

- Clients: sequential processes
  - ⋆ Infinite arrival process with finite concurrency
  - ⋆ Each client has a distinct identity
  - ⋆ Crash failure model (Recovery with a new id)
  - ⋆ Wait-free
- Shared object
  - ⋆ Read/write operations
  - ⋆ Correctness criterion: Linearizability

Shared memory: set of servers

- Distributed message-passing system made up of servers
- Infinite arrival model with finite concurrency
- Server s_j can:
  - ⋆ Enter the system (event $init_j$)
  - ⋆ Crash (event $fail_j$) or leave (event $leave_j$)
- Each object: implemented by dynamic subset of servers
- Notation: $up(\tau)$= the servers (implementing object x that have entered the system before time τ, and have neither crashed or left before τ
- Feasability condition: $\forall\tau$: $up(\tau) \neq \emptyset$

Shared memory



Looking for Appropriate Abstractions

- If servers enter and leave the system arbitrarily fast: nothing can be done
- Any dynamic system requires some form of eventual stability "during long enough periods" in order non-trivial computations can progress
- Here we consider $\boxed{\text{abstract properties}}$ (instead of particular duration assumptions)
  - ⋆ Similarly to the failure detector approach, these properties are not related to specific synchrony or duration assumptions. This favors good software engineering practice (modularity, portability, proof)
  - ⋆ Two Abstractions
    - ∗ Read/Write dynamic quorums
    - ∗ Persistent reliable broadcast

Operations as Intervals

- The ath execution of a read or write operation by a client p_i defines an interval I_i^a
- A run (or history h) is a totally ordered sequence of the events issued by the clients
- Partial order on intervals:
  - ⋆ $I1 \rightarrow_h I2$ if the last event of $I1$ precedes in h the first event of $I2$
  - ⋆ $im\_pred(I1, I2)$ (immediate predecessor)

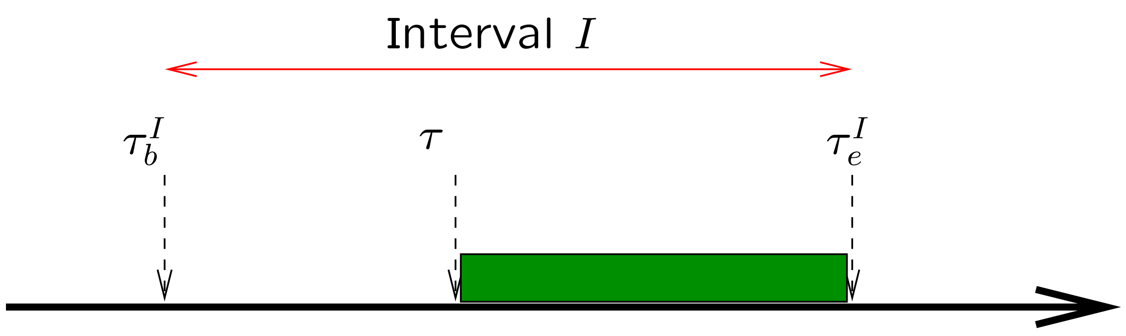
$$\text{if } I1 \rightarrow_h I2 \text{ and } \nexists I : \; I1 \rightarrow_h I \wedge \; I \rightarrow_h I2$$

Associating a stability set with each interval (1)

- I an interval that starts at time τ_b^I and ends at time τ_e^I
- The following set of servers is assoc. with each inter. I:

$$STABLE(I) =$$

$$\boxed{\{s \mid \exists \tau \in [\tau_b^I, \tau_e^I] : \quad \forall \tau' : \; \tau \leq \tau' \leq \tau_e^I : \quad s \in up(\tau') \; \}}$$



Associating a stability set with each interval (2)

- Feasibility condition necessary to obtain live quorums:

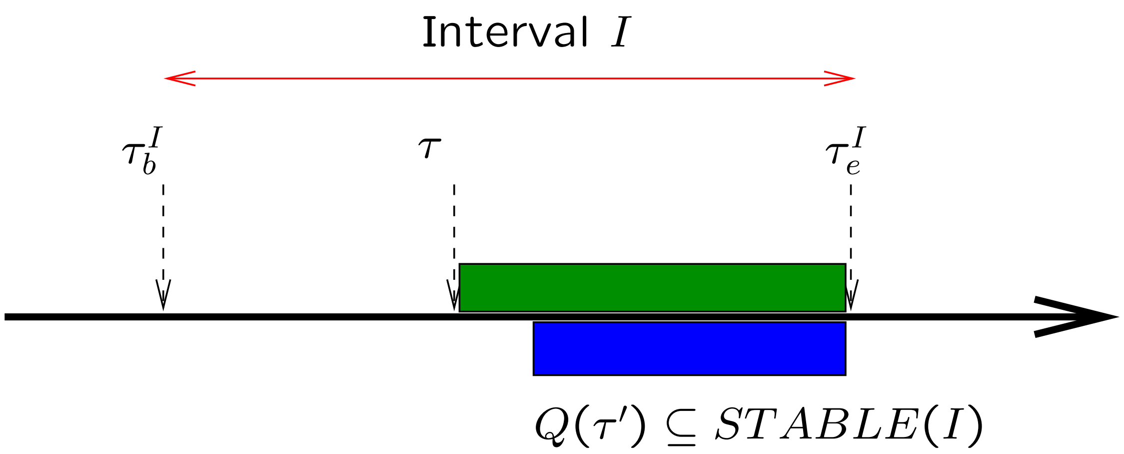
$$\boxed{\forall I : \quad STABLE(I) \neq \emptyset}$$

Dynamic Read/Write Quorums (1)

- Let $Q(t)$ be the quorum (set of servers) returned by a quorum query issued at time t during an interval I

- Progress property:

$$\exists \tau \in [\tau_b^I, \tau_e^I] : \quad \forall \tau' : \tau \leq \tau' \leq \tau_e^I : \; Q(t') \subseteq STABLE(I)$$



- This means that an operation can eventually obtain a quorum of alive servers: this property is a requirement to ensure the liveness of read and write operations

Dynamic Read/Write Quorums (2)

- A read/write op can invoke two types of quorums:
  - ⋆ cd query: to obtain a control data
  - ⋆ val query: to obtain a data

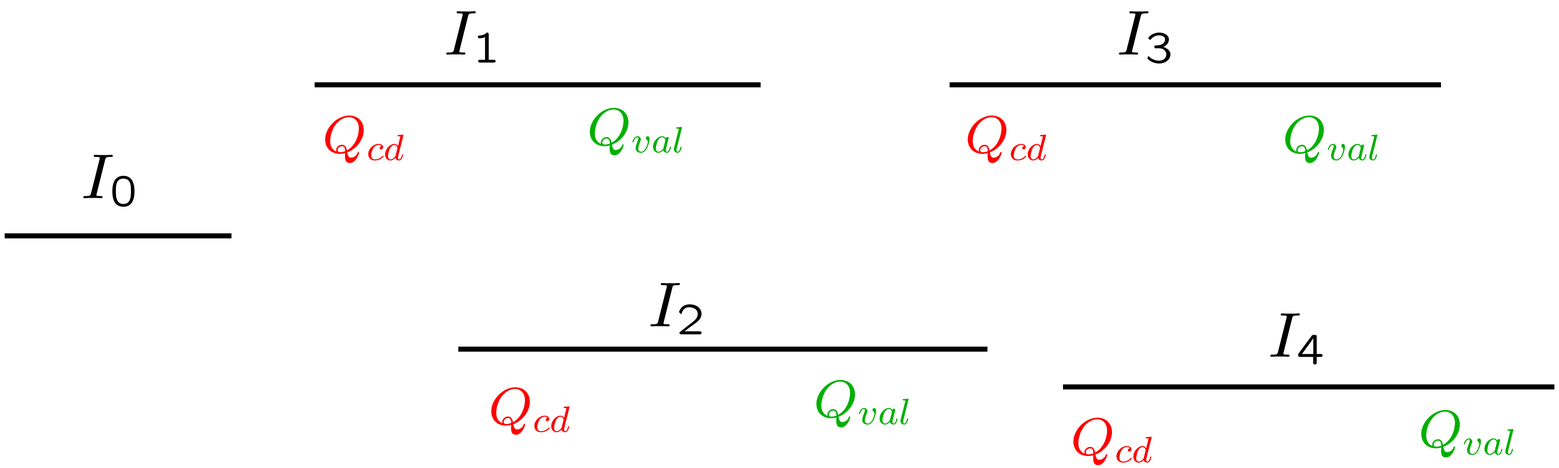
- Typed Bounded Lifetime Intersection property:

$$(Q_{val} \in I1) \;\wedge\; (Q_{cd} \in I2) \;\wedge\; im\_pred(I1, I2)$$
$$\Rightarrow Q_{val} \cap Q_{cd} \neq \emptyset$$

It is not required that any pair of quorums intersect

It is not required that any pair of consecutive or concurrent quorums intersect

Intervals



$$
\begin{tikzcd}
 & I_1 \arrow[r] \arrow[ddr] & I_3 \\
I_0 \arrow[ur] \arrow[dr] & & \\
 & I_2 \arrow[r] & I_4
\end{tikzcd}
$$

$$Q_{val}(I1) \cap Q_{cd}(I3) \neq \emptyset$$

$$Q_{val}(I2) \cap Q_{cd}(I4) \neq \emptyset$$

$$Q_{val}(I1) \cap Q_{cd}(I4) \neq \emptyset$$

Persistent Reliable Broadcast (1)

- Extend Uniform Reliable Broadcast to dynamic systems
- Notion of persistence in message delivery
- Two primitives: prst_broadcast(m) and prst_deliver()
- Each message m has a type $type(m)$ and a sequence number $sn(m)$
- Defined by four properties:
  - ⋆ Validity: If a message m is delivered by a server, it has been broadcast by a read or write operation
  - ⋆ Uniformity: A message m is delivered at most once by a server

Persistent Reliable Broadcast (2)

- Server/server Termination: If a message m, broadcast during an interval I, is delivered by a server, then any server $s \in STABLE(I)$ eventually delivers a message m' such that $type(m) = type(m')$ and $sn(m') \geq sn(m)$

- Client/server Termination: If the client process does not crash while it is executing the read or write operation defining the interval I that gave rise to the broadcast of m, the message m is delivered by at least one server

Implementing an Atomic Object Service

- Associate a timestamp with each value (classical)
- A read or write operation: two steps [ABD 1995]
  - ⋆ Phase 1: Acquire the "last" timestamp
  - ⋆ Phase 2: Ensure consistency of the read/write op
- Here we present only the write operation (read is similar)

Implementing a WRITE operation (1)

operation write$_i$ (x, v)

% Phase 1: synchro to obtain consistent information %
$sn_i \leftarrow sn_i + 1$; $ans_i \leftarrow \emptyset$;
$\boxed{\text{prst\_broadcast } \textit{cd\_req}(i, sn_i, \text{no})}$;
repeat
  wait for a message *cd_ack*(sn_i, ts) received from s;
  $ans_i \leftarrow ans_i \cup \{s\}$
until $\boxed{(Q_{cd} \subseteq ans_i)}$;
$ts.clock \leftarrow$ max of the $ts.clock$ fields received $+1$;
$ts.proc \leftarrow i$;

Implementing a WRITE operation (2)

% Phase 2 : synchro to ensure atomic consistency %

$\boxed{\text{prst\_broadcast } \textit{write\_req}(i, sn_i, ts, v)}$;

$ans_i \leftarrow \emptyset$;

repeat

wait for a message *write_ack*(sn_i) received from s;

$ans_i \leftarrow ans_i \cup \{s\}$

until $\boxed{(Q_{val} \subseteq ans_i)}$;

return()

Implementing on the Sever side

Server s maintains the value $value_s$ whose timestamp is ts_s

when *cd_req*$(i, sn, bool)$ is **delivered**:
if $(bool$ =yes)
then $val\_to\_send \leftarrow value_s$
else $val\_to\_send \leftarrow \perp$
end_if;
send *cd_ack*(sn, ts, val_to_send) *to* i

when *write_req*(i, sn, ts, v) is **delivered**:
if $(ts > ts_s)$ **then** $ts_s \leftarrow ts$; $value_s \leftarrow v$ **end_if**;
send *write_ack*(sn) *to* i

Proof (1)

- Theorem Read and write liveness

  A read or write operation executed by a process p_i that does not crash terminates

  The proof relies on the stability condition

- Definitions:

  ⋆ Let an effective write be a write operation whose request has been delivered by at least one server

  Let $ts(w)$ be the timestamp associated with the effective write operation w

  ⋆ Let an effective read be a read operation that does not crash

  Let $ts(r)$ be the timestamp associated with the effective read operation r

Proof (2)

- Theorem Timestamp ordering property

  Let $op1$ and $op2$ be two effective operations, $I1$ and $I2$ their intervals with $I1 \rightarrow_h I2$. We have:

  (i) If $op1$ is a read or a write operation and $op2$ is a read operation then $ts(op1) \leq ts(op2)$

  (ii) If $op1$ is a read or a write operation and $op2$ is a write operation then $ts(op1) < ts(op2)$

- Theorem Atomic consistency

  There is a total order on all the effective operations that (1) respects their real-time occurrence order, and (2) such that any read operation obtains the value written by the last write that precedes it in this sequence

A few works on quorums (quora!) in dynamic systems

A (really) non-exhaustive list

- Herlihy, M., Dynamic Quorum Adjustment for Partitioned Data. *ACM Transactions on Database Systems*, 12(2):170-194, 1987
- Lynch, N. and Shvartsman A., RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 173-190, 2002
- Nadav U. and Naor M., The Dynamic And-or Quorum System. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag LNCS #3724:472-486, 2005
- Abraham, I. and Malkhi, D., Probabilistic Quorum Systems for Dynamic Systems. *Distributed Computing*, 18(2):113-124, 2005.
- Gramoli V. and Raynal M., Timed Quorum Systems for Large-Scale and Dynamic Environments. *Proc. 11th Int'l Conf. on Principles of Distributed Systems (OPODIS'07)*, Springer-Verlag, LNCS # 4878:429-442, 2007

Implementing a regular register in a dynamic system

- Implementing a Register in a Dynamic Distributed System. *IEEE Int'l Conf. on Distributed Computing Systems (ICDCS'09)*, Montréal (Canada), 2009 (Baldoni R., Bonomi S., Kermarrec A.-M. and Raynal M.)

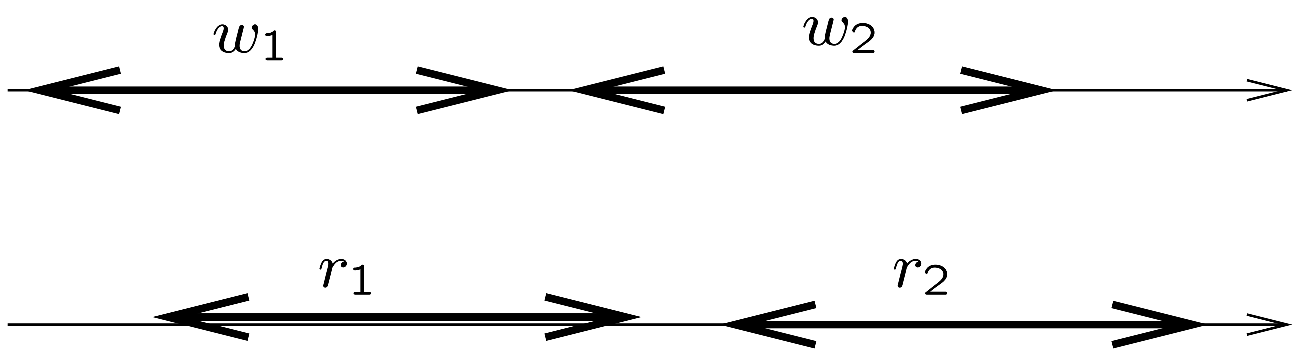
Regular register

- It is a very basic (primitive) object
- Assumption: no two writes are concurrent (hence the notion of last write is well-defined)
- Safety: A read returns the last value written before the read invocation or a value written by a concurrent write
- Liveness: If a process invokes a read or write operation and does not leave the system, it returns from that operation

- Lamport. L., On Interprocess Communication: Part 1: Models, Part 2: Algorithms. *Distributed. Computing.*, 1(2):77-101, 1986

- Shao C., Pierce E. and Welch J., Multi-writer consistency conditions for shared memory objects. *Proc. DISC'03,* LNCS #2848:106-120, 2003

The New/old inversion phenomenon



Related work

- Some works require that a correct process remain forever in the system

  Here any process can be replaced at any time according to the churn

- Works on MANET:

  - ⋆ Do not consider the churn parameter
  - ⋆ Based on a deterministic broadcast op (GeoCast)

- Dolev S., Gilbert S., Lynch N., Shvartsman A., and Welch J., Geoquorum: Implementing Atomic Memory in Ad hoc Networks. *Proc. DISC'03*, LNCS #2848:306-320, 2003

- Roy M., Bonnet F., Querzoni L., Bonomi S., Killijian M.O. and Powell D., Geo-Registers: an Abstraction for Spatial-Based Distributed Computing. *Proc. OPODIS'08*, LNCS #5401:354-357, 2008

Synchronous dynamic system (1)

- Infinite arrival model: a run can have an infinite nb of processes, but this nb is finite in any finite period
- Each process has its own id
- A process can re-enter with a new id (new process)
- A process first issue a join(), then it accesses the register and possibly leave the system
- Churn rate $c \in [0..1]$ (percentage of the processes that are "refreshed" per time unit): constant

  - Ko S., Hoque I. and Gupta I., Using Tractable and Realistic Churn Models to Analyze Quiescence Behavior of Distributed Protocols. *Proc. 27th IEEE Int'l Symposium on Reliable Distributed Systems (SRDS'08)*, pp. 259-268, 2008

Synchronous dynamic system (2)

- Local computation: no duration
- There is an upper bound δ for 1-to-1 and 1-to-* message transfer delays
- A process that enters the system becomes active only after some duration (δ or 3δ)

  During that period it can receive and process messages

Read operation

- Local variables
  - $\star$ $register_i$
  - $\star$ sn_i
- Fast read operation
  operation read(): return($register_i$).

Write operation

operation write(v): % issued only by the writer p_w%

$sn_w \leftarrow sn_w + 1$; $register_i \leftarrow v$ broadcast write(v, sn_w);

wait(δ); return(ok).

when write($< val, sn >$) is delivered: % at any process p_i %

if $(sn > sn_i)$ **then** $register_i \leftarrow val$; $sn_i \leftarrow sn$ **end if**.

Join operation

- Local variables
  - ⋆ $active_i$: Boolean that becomes true at the end of the join() operation (p_i becomes active)
  - ⋆ $replies_i$: a set that keeps the replies $< j, value, sn >$ received by p_i during its waiting period (while it is joining)
  - ⋆ $reply\_to_i$: a set that keeps the set of processes that are currently joining and asked p_i to send them its current sate $< i, register_i, sn_i >$ (as soon as it is active)

Join operation (1)

operation join(i):

$register_i \leftarrow \perp$; $sn_i \leftarrow -1$; $active\_i \leftarrow false$;

$replies_i \leftarrow \emptyset$; $reply\_to_i \leftarrow \emptyset$;

wait(δ);

if ($register_i = \perp$) **then**

  $replies_i \leftarrow \emptyset$;

  broadcast inquiry(i);

  wait(2δ);

  let $< id, val, sn > \in replies_i$ **such that**

    $(\forall < -, -, sn' > \in replies_i : \; sn \geq sn')$;

  if ($sn > sn_i$) **then** $sn_i \leftarrow sn$; $register_i \leftarrow val$ **end if**

end if;

$active_i \leftarrow true$;

for each $j \in reply\_to_i$ **do** send reply $(< i, register_i, sn_i >)$ to p_j;

return(ok).

Join operation (2)

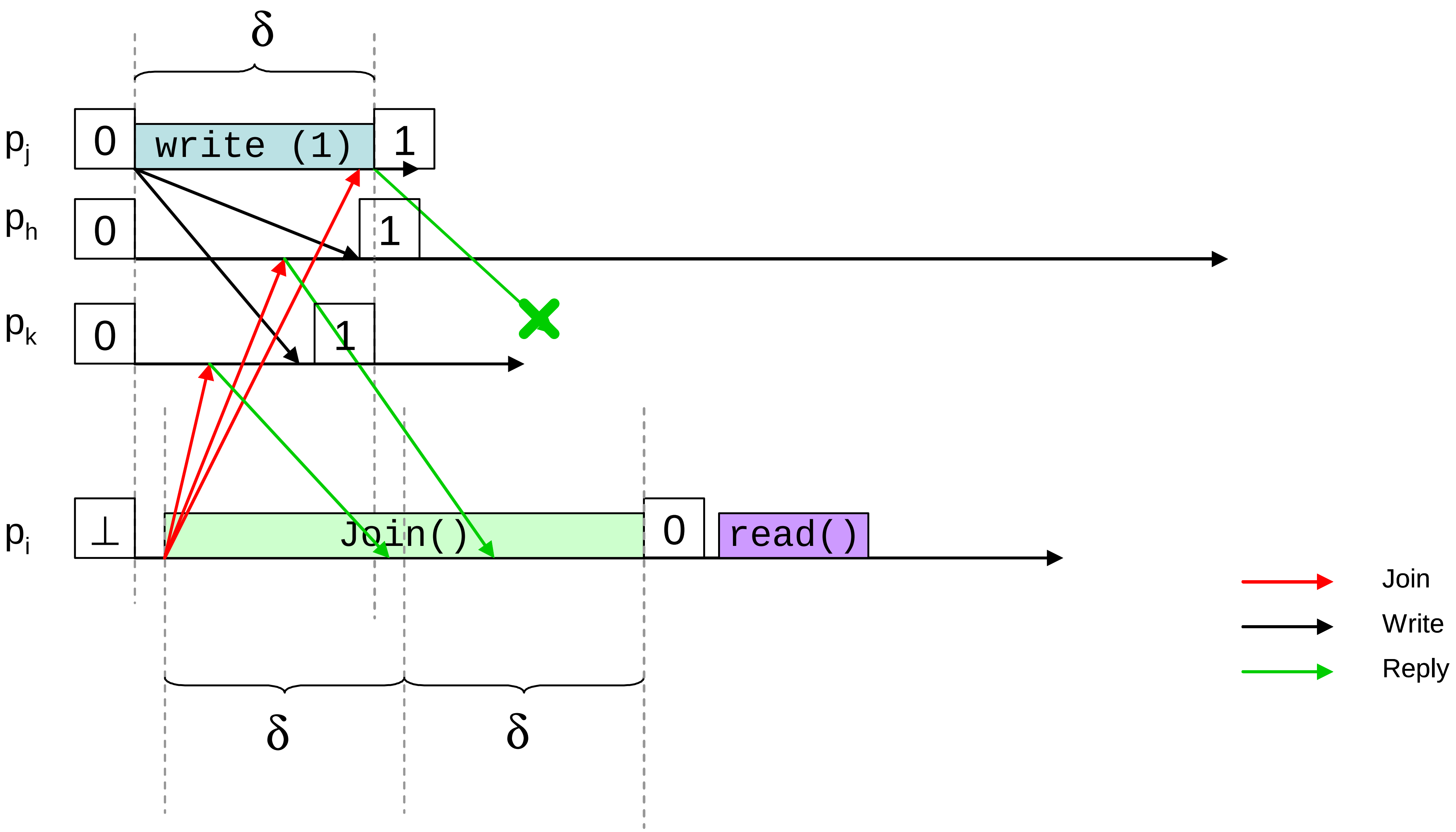
when inquiry(j) is delivered:
if $(active_i)$ **then** send reply $(< i, register_i, sn_i >)$ to p_j
else $reply\_to_i \leftarrow reply\_to_i \cup \{j\}$
end if.

when reply($< j, value, sn >$) is received:
$replies_i \leftarrow replies_i \cup \{< j, value, sn >\}$.

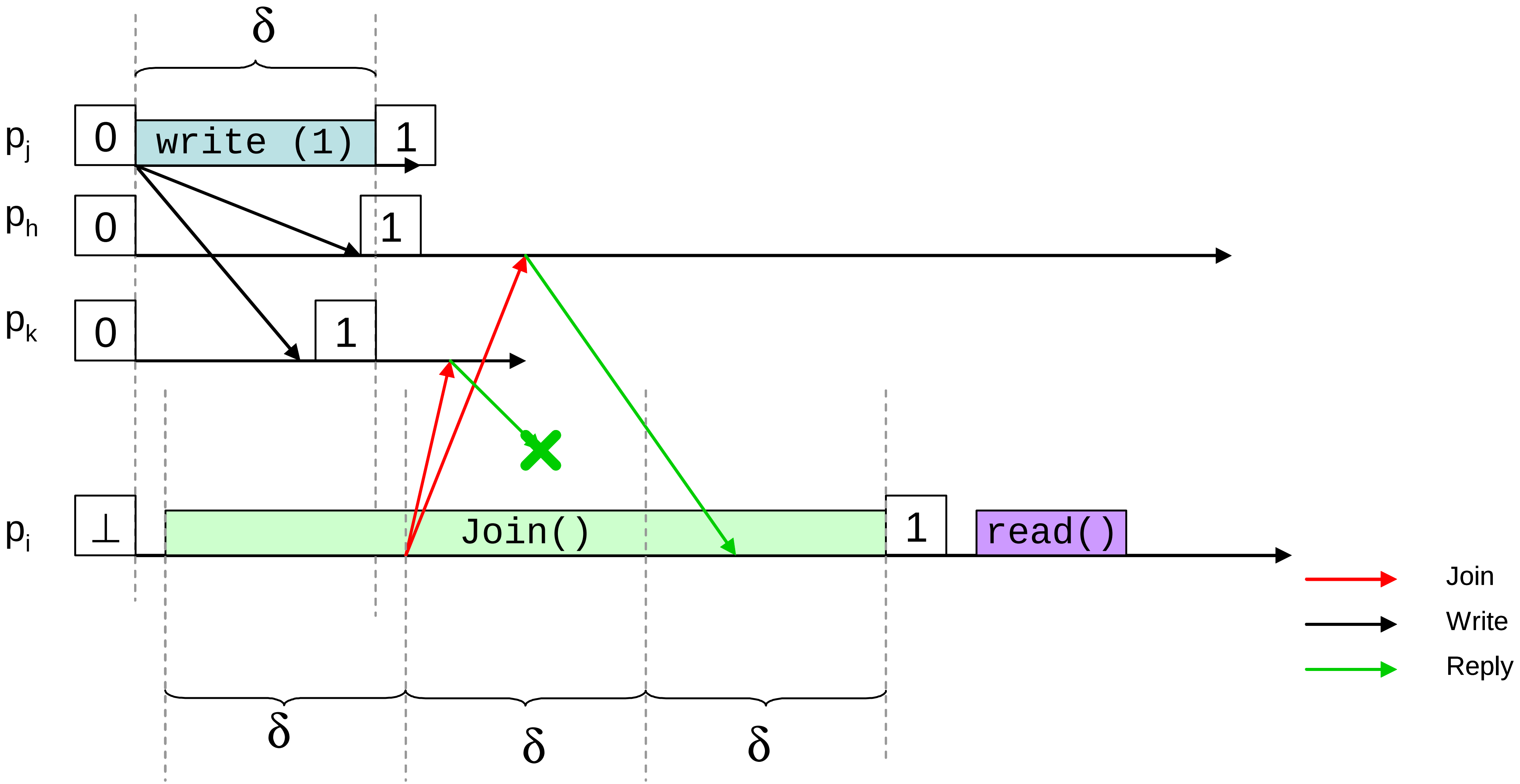
Remark:
For people familiar with Ricart-Agrawala 1981 Mutex algorithm: the proposed algorithm is very close...

More personal remark:
I do not think that there are plenty of basic principles and basic mechanisms...
Computer science is a science of abstraction...

Why "wait(δ)"?



wait(δ): stabilization period



Results

- Theorem: If $c < 1/(3\delta)$ the algorithm implements a regular register in a synchronous distributed system
- Theorem: Such a construction is impossible in a pure asynchronous dynamic system
- Theorem: An algorithm exists in eventually synchronous dynamic systems when
  - $\star$ $c < 1/(3\delta n)$, and
    $\forall\ \tau : |A(\tau)| > n/2$
    ($A(\tau)$ is the set of processes $active$ at time τ)

Open problems (among others!)

- Is it possible to characterize the greatest value of c for a synchr system (as a function involving the bound δ)?
- Has c to depend on n in an eventually synch system?
- How to cope with the effect of dynamicity + failures?

The only slide to remember

- Dynamicity does modify our view of what is a system
- The important concept is Adaptability
- Today, we only have a few disparate elements, and are only in the infancy of dynamic systems (lots of applications, no strong theory)
- A global view and the associated underlying concepts are still to be discovered

Self-organizing systems

Large scale networked systems: from anarchy to geometric self-structuring. *Proc. 10th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer Verlag LNCS #5408, pp. 25-36, 2009 (Kermarrec A.-M., Mostefaoui A., Raynal M., Viana A., Trédan G.)

Self-structuring

- Self-structuring represents the abilty of a system to let emerge a specific structure from scratch without requiring external intervention

- A key feature of autonomy

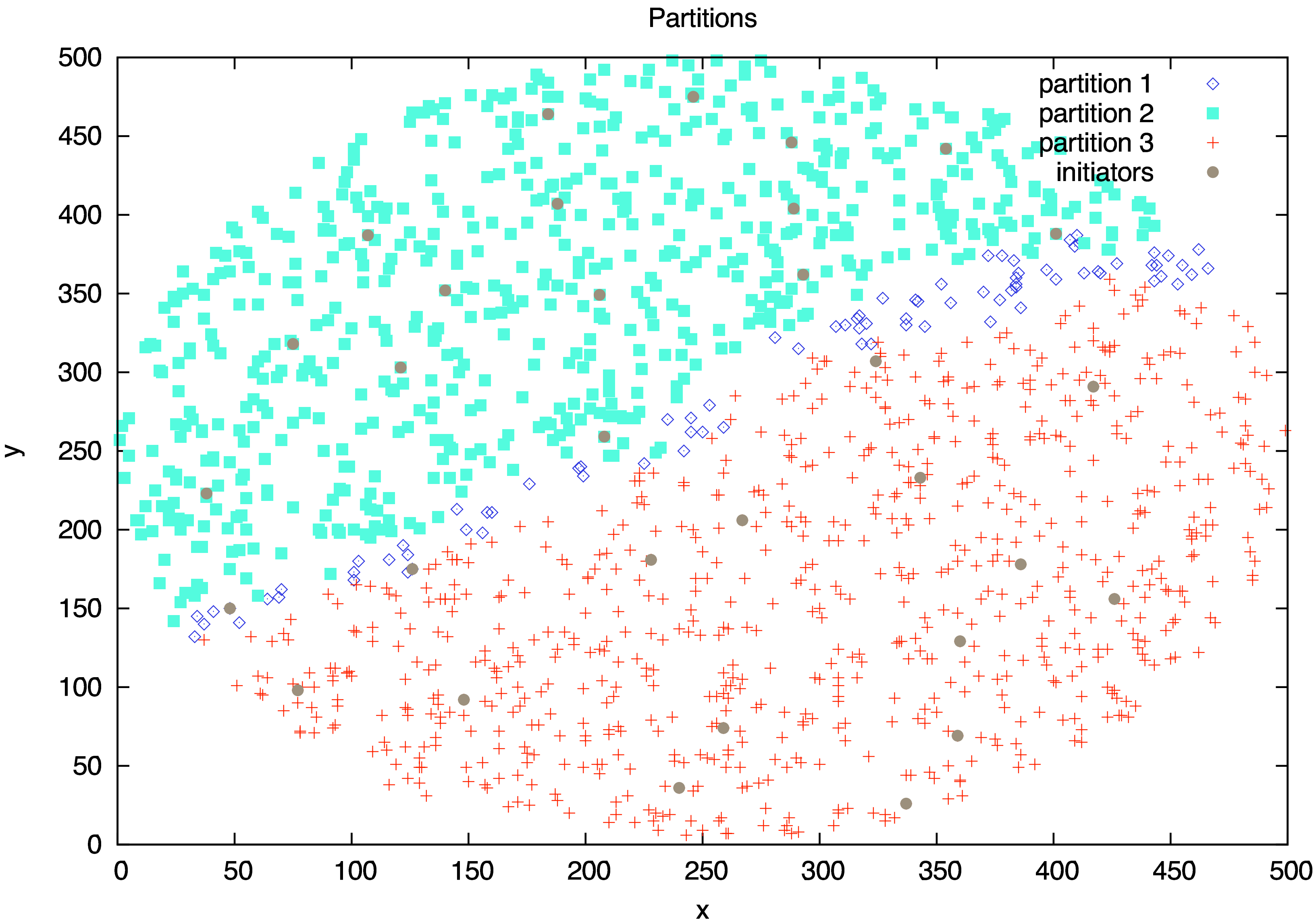
- In sensor networks: self-structuring represents an important requirement for operations such as forwarding, load balancing, leader election energy consumption management, etc.

- Example: Partitioning into several zones for monitoring purposes, or selection of sensors to ensure specific functions (and save energy)

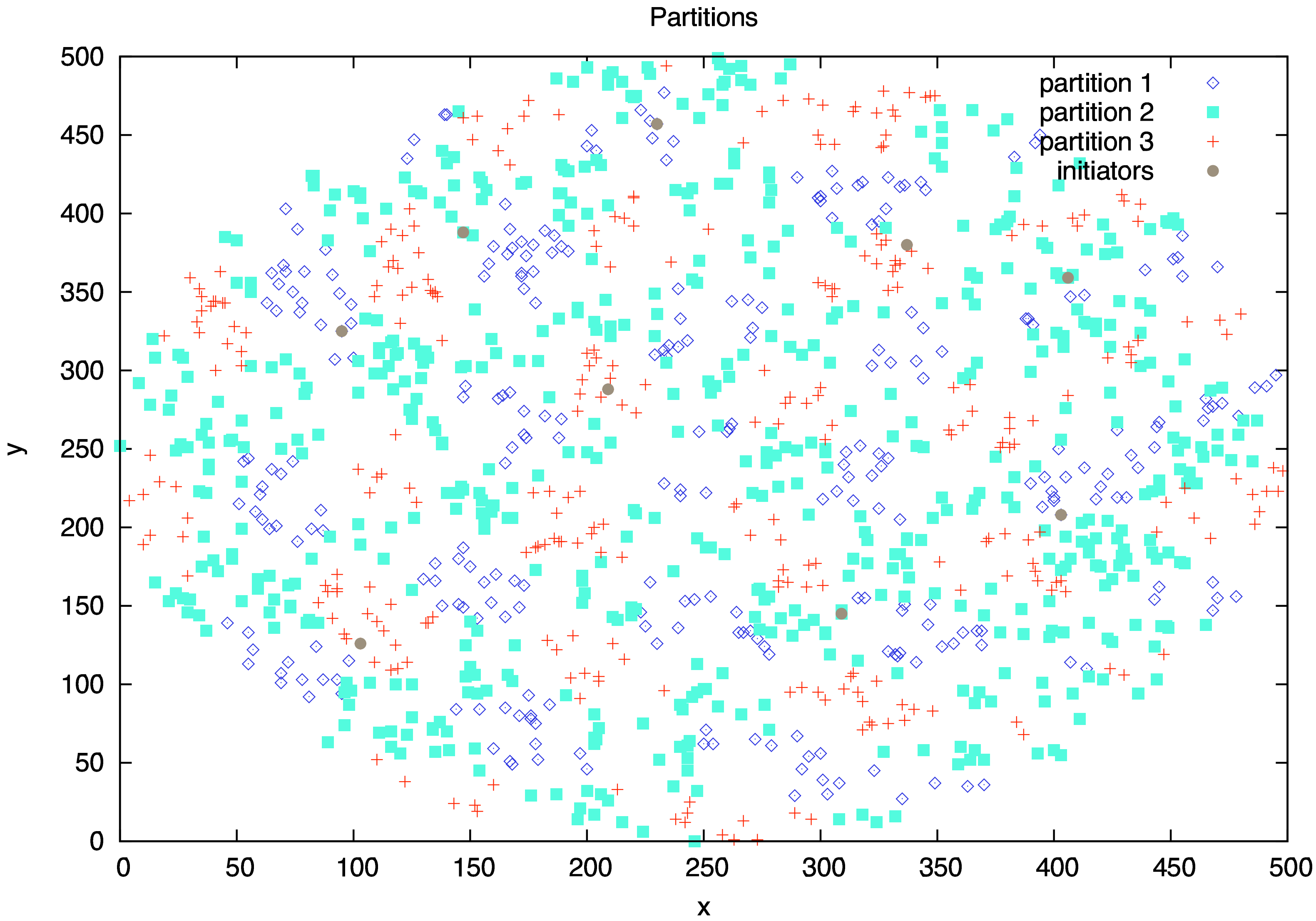
Network organization: Geometric structuring

- A network organization is based on an underlying structure that is geographical or functional
- Geographical example: sector-shaped clustering
- Functional example: awake and sleep entities

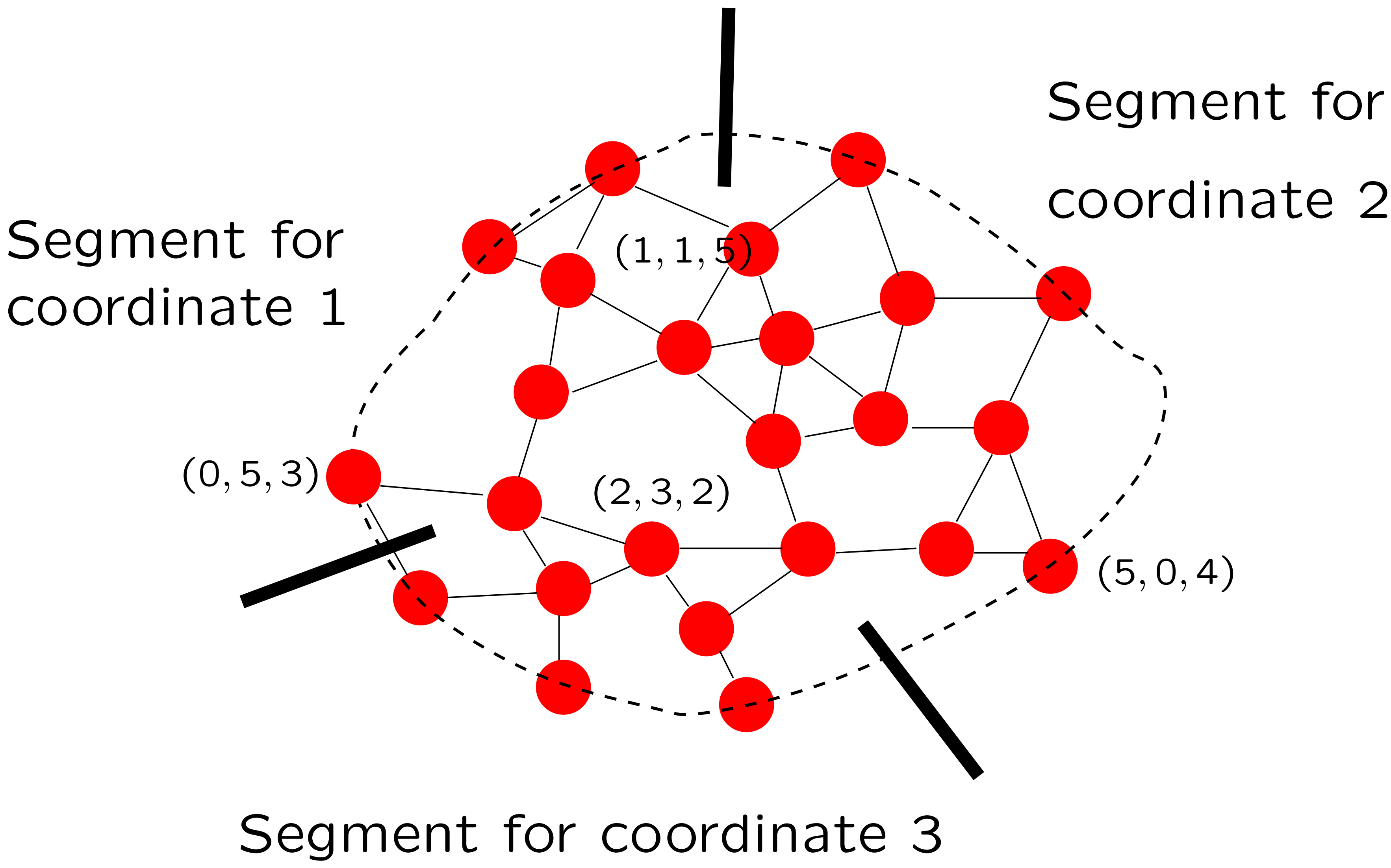
Example 1: North, South, and Equator zones



Example 2: Wake up/Sleep entities



Virtual coordinates: illustration



Underlying rationale

- Not directly related to "real geographic coordinates"
- $\boxed{\text{Connectivity-based approach}}$: the coordinates reflect only the underlying connectivity
- Can adapt to obstacles (mountains, underground, etc.)

Using VC for network structuring

- Geometric/Functional structuring
- Aim: associate a partition number p with each entity
- Use a mathematical function to define a structure
  - ⋆ Let an entity i, with coordinate c_i
  - ⋆ The function $f()$

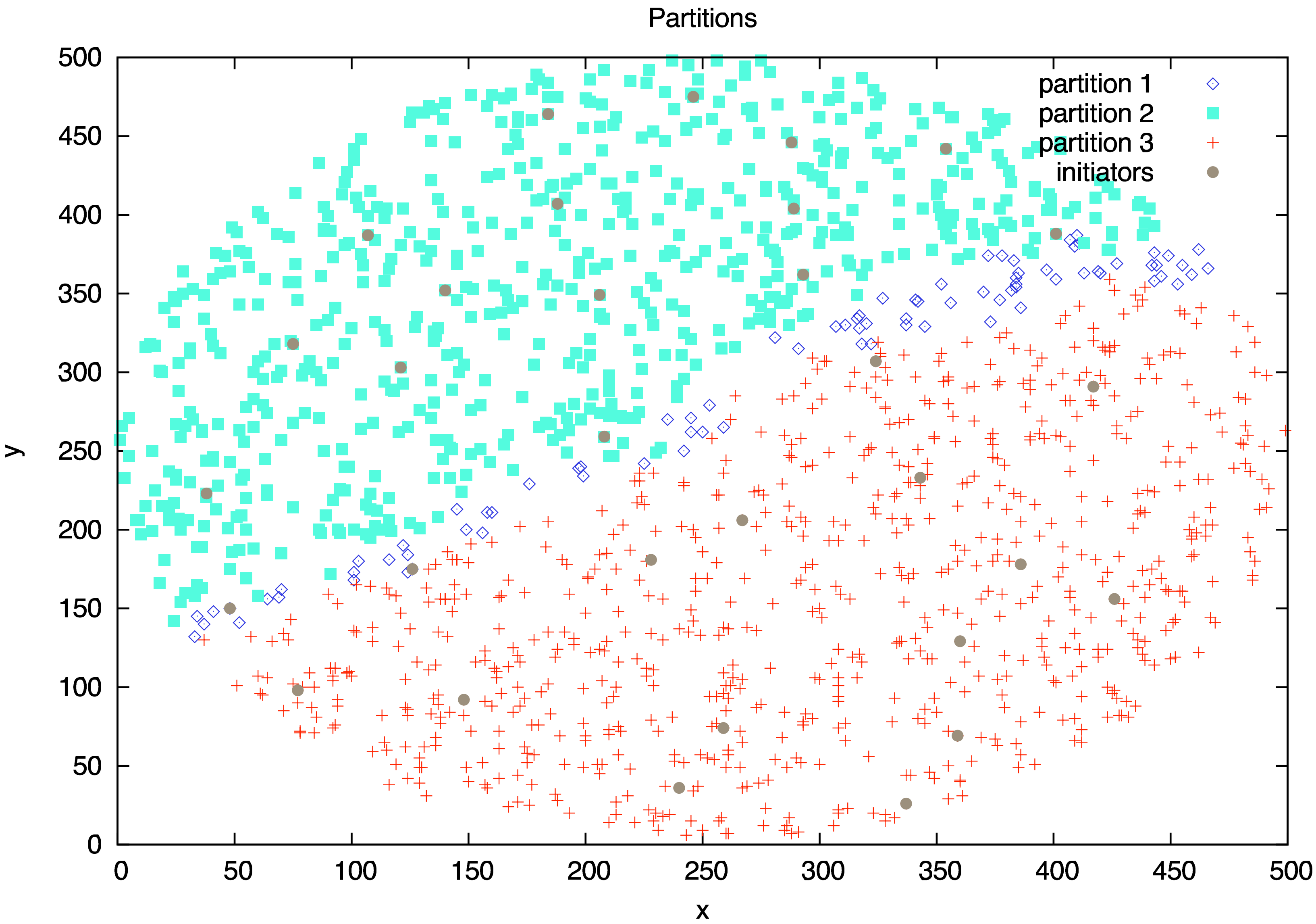
$$\begin{aligned} f : \mathcal{K} &\rightarrow \{0, \ldots, p\}, \\ f(c_i) &\rightarrow p_i. \end{aligned}$$

Geometric structuring (1.1)

- North, South and Equator partitioning
- $d = 2$, $p = 3$
- The function $f()$

$$
\begin{array}{rcl}
f : \mathbb{N} * \mathbb{N} & \rightarrow & \{1, 2, 3\} \\
 & & 1 \quad when \quad x_1 > x_2 \\
f(x_1, x_2) & \rightarrow & 2 \quad when \quad x_1 = x_2 \\
 & & 3 \quad when \quad x_1 < x_2
\end{array}
$$

Geometric structuring (1.2)

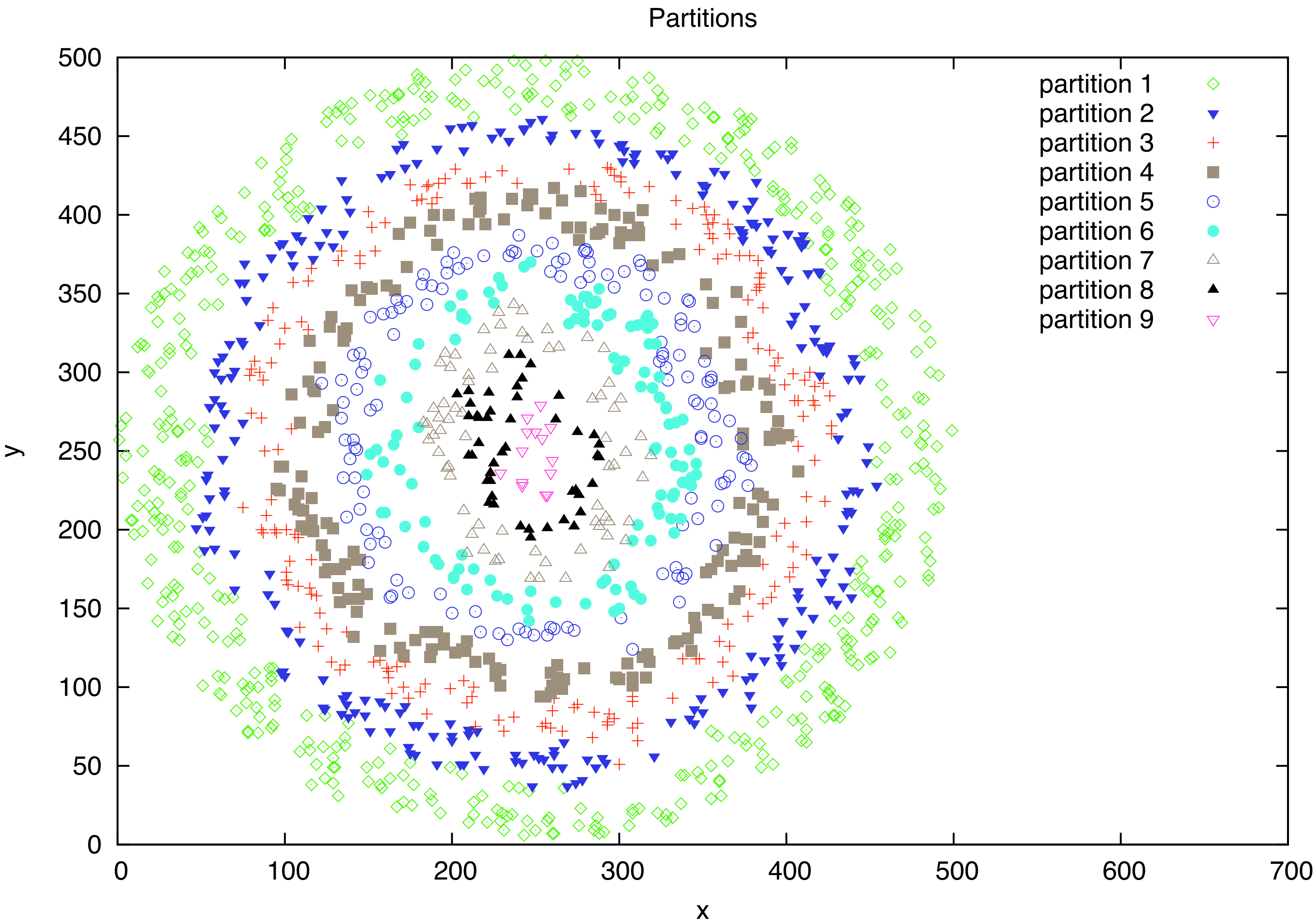


Geometric structuring (2.1)

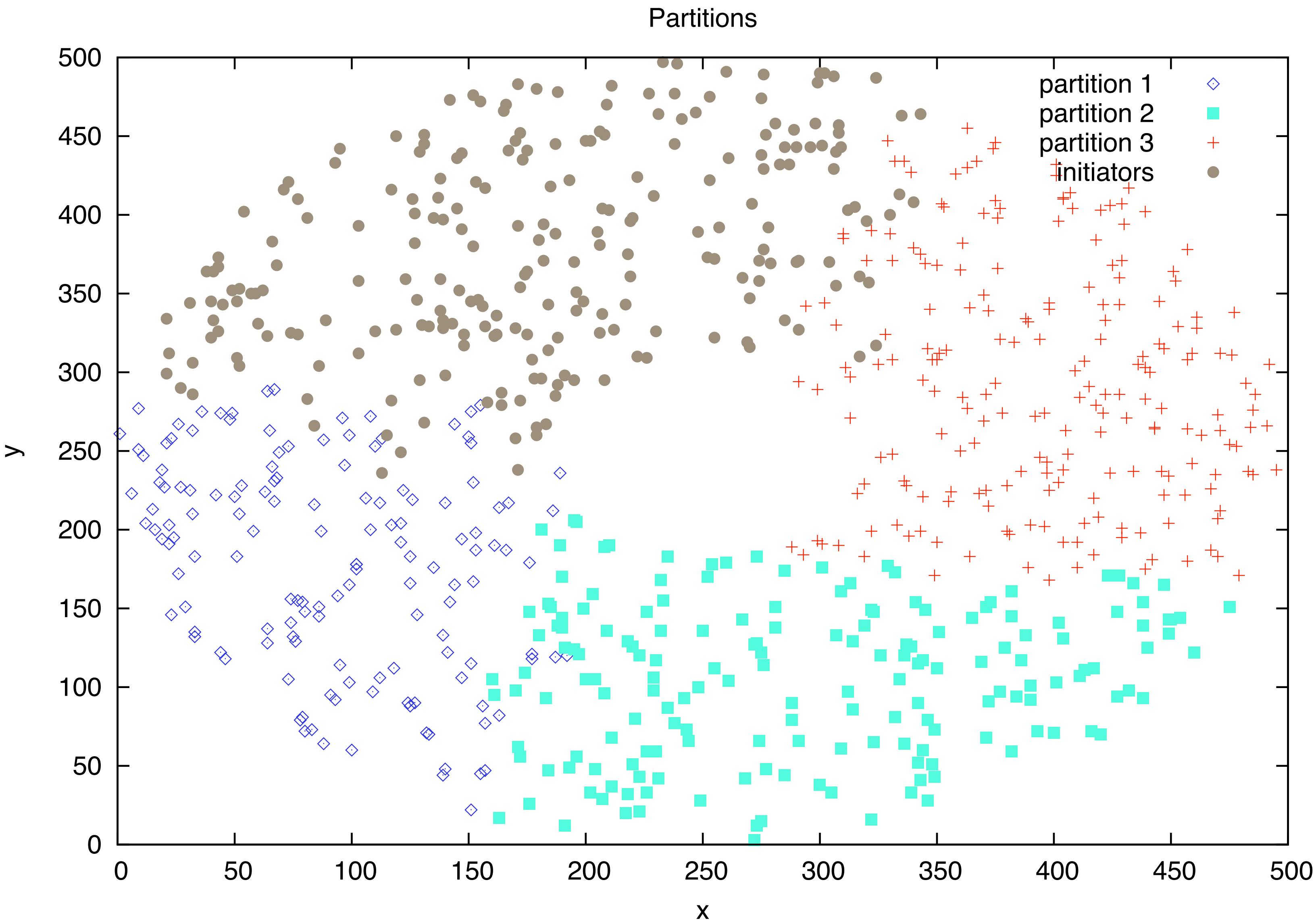
- Target partitioning
- $d = 1$, $p > 0$
- "Wave" waking up
- The function $f()$

$$\begin{aligned} f : \mathbb{N} &\rightarrow \mathbb{N} \\ f(x_1) &\rightarrow x_1 \end{aligned}$$

Geometric structuring (2.2)

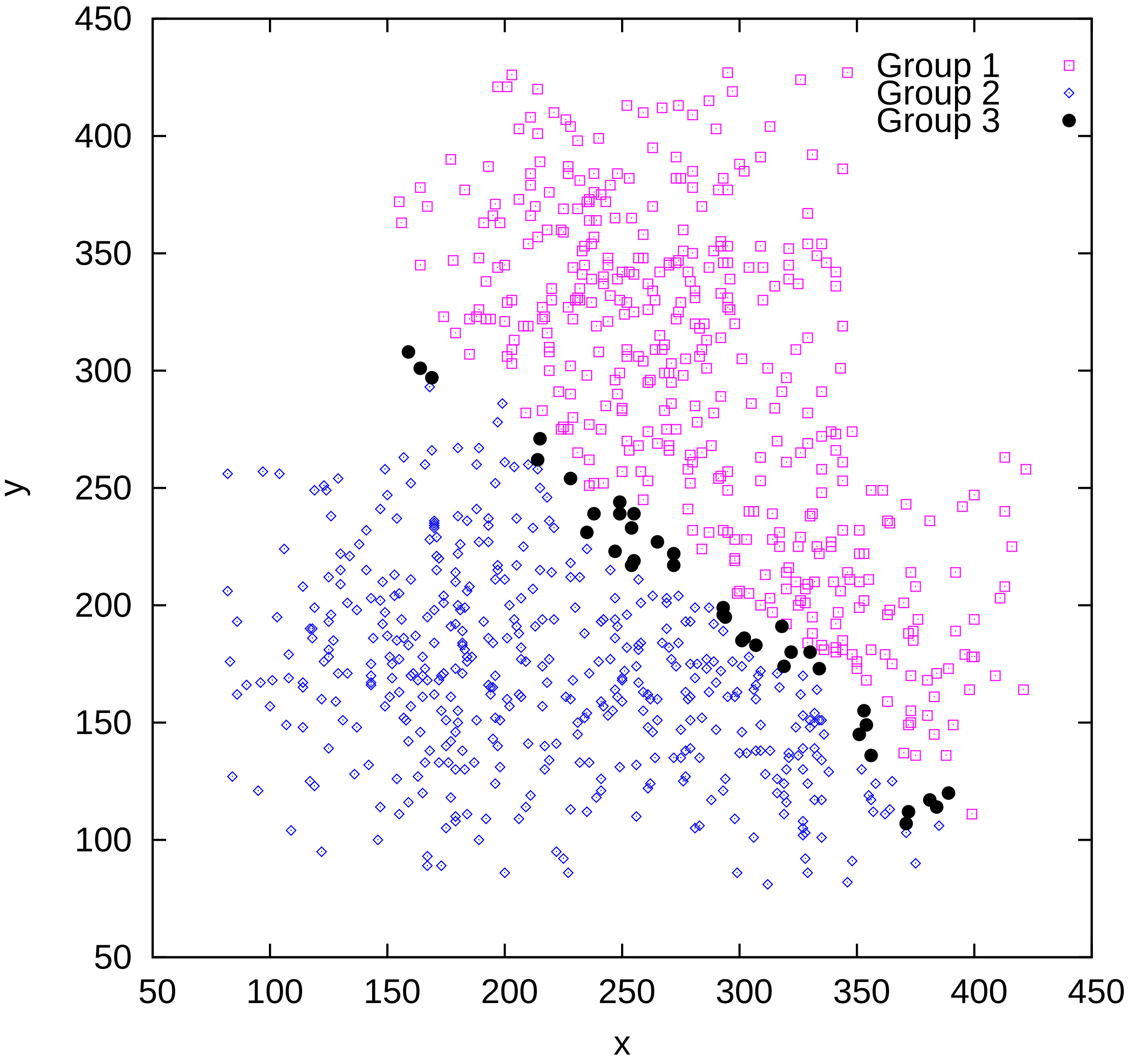


Functional structuring (3): with a hole at the center



Functional structuring (4)

The “line” predicate with three hot spots of different node densities



What we have seen

- From a static to a dynamic system
- Atomic objects in a dynamic system
- Scalability: geometric structuring
- implementation of a regular register in a dynamic system the churn of which
  - ⋆ The churn is fixed and known
  - ⋆ The churn can vary within some known range
- Distributed slicing (locality)

The only slide to remember

- Dynamicity does modify our view of what is a system
- The important concept is Adaptability
- Today, we only have a few disparate elements, and are only in the infancy of dynamic systems (lots of applications, no strong theory)
- A global view and the associated underlying concepts are still to be discovered