Abstractions for Implementing Atomic Objects in Dynamic systems

R. FRIEDMAN† M. RAYNAL⋆ C. TRAVERS⋆

roy@cs.technion.ac.il {raynal|ctravers}@irisa.fr

†Technion, Haifa, Israel

⋆IRISA, Université de Rennes, France

Summary

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

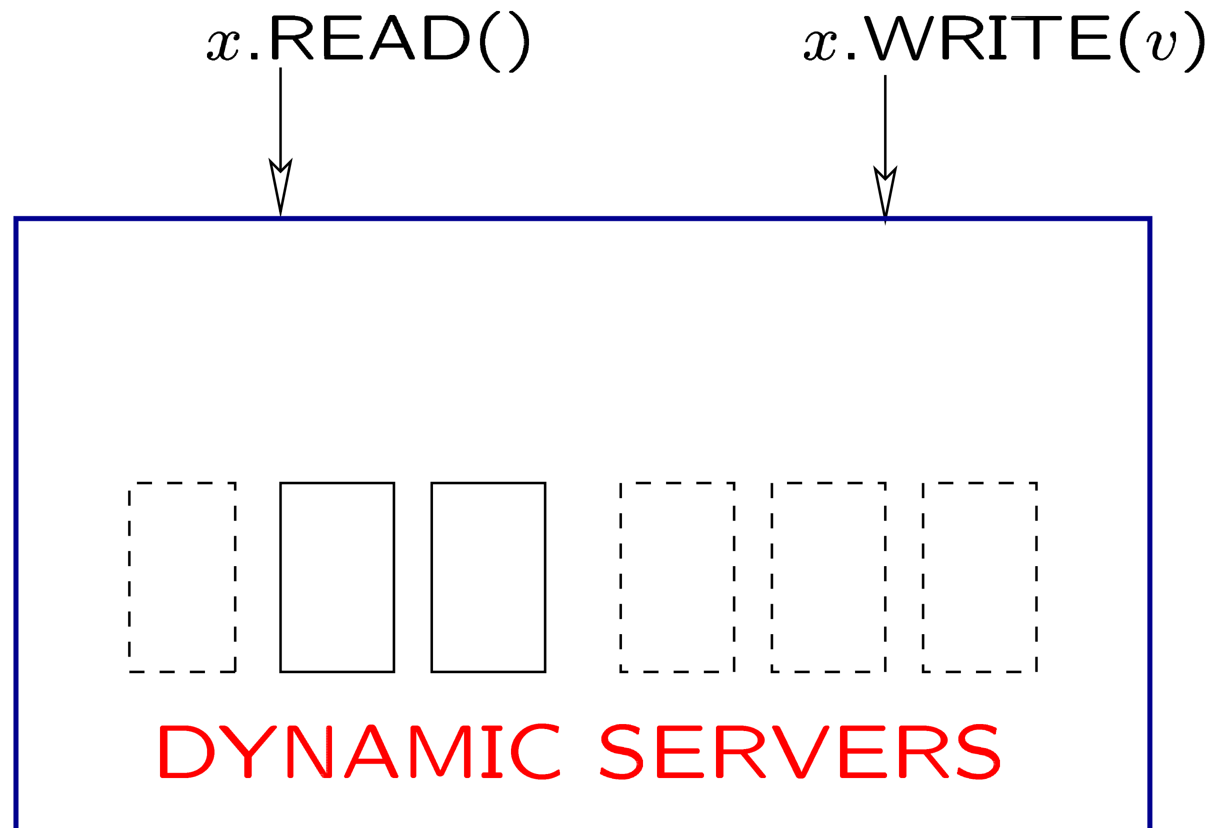
Dynamic systems

- 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: $up(t)$= the servers (implementing object x that have entered the system before time t and have neither crashed or left before t
- Feasability condition: $\forall t$: $up(t) \neq \emptyset$

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 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:
  - $\star$ $I1 \rightarrow_h I2$ if the last event of $I1$ precedes in h the first event of $I2$
  - $\star$ $im_pred(I1, I2)$ (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

- I an interval that starts at time t_b^I and ends at time t_e^I
- The following set of servers is associated with each interval I:

$$STABLE(I) =$$

$$\boxed{\{s \mid \exists t \in [t_b^I, t_e^I] : \quad \forall t' : \ t \leq t' \leq t_e^I : \quad s \in up(t') \ \}}$$

- 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:

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

- 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:

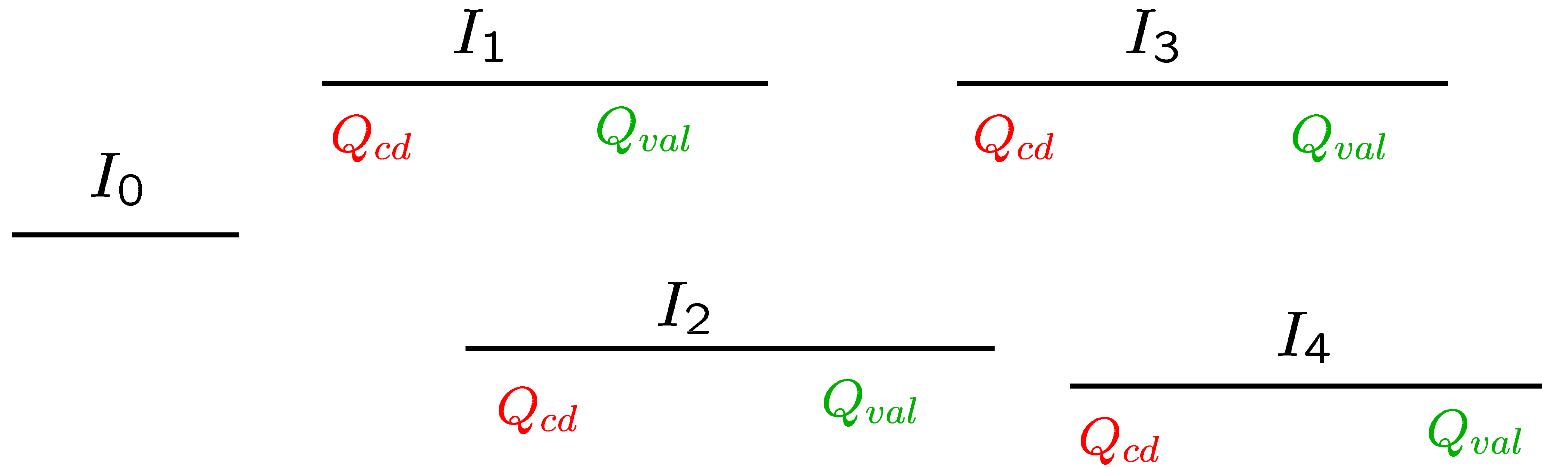
$$(Q_{val} \in I1) \ \wedge \ (Q_{cd} \in I2) \ \wedge \ im_pred(I1, I2)$$

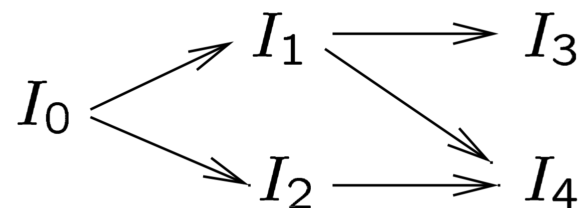
$$\Rightarrow Q_{val} \cap Q_{cd} \neq \ \emptyset$$

It is not required that any pair of quorums intersect

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

Intervals





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

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

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

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 [Attiya-Bar Noy-Dolev 1995]
  - ⋆ Phase 1: Acquire the “last” timestamp
  - ⋆ Phase 2: Ensure consistency of the read or write operation
- 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 %

$sn_i \leftarrow sn_i + 1$; $ans_i \leftarrow \emptyset$;

prst_broadcast *cd_req*(i, sn_i, no);

repeat

wait for a message *cd_ack*(sn_i, ts) received from s;

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

until $(Q_{cd} \subseteq ans_i)$;

$ts.clock \leftarrow$ max of the $ts.clock$ fields received $+1$;

$ts.proc \leftarrow i$;

Implementing a WRITE operation (2)

% Phase 2 : synchro to ensure atomic consistency %

prst_broadcast *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 $(Q_{val} \subseteq ans_i)$;
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

Conclusion

- Two new abstractions for dynamic systems
  - ⋆ Dynamic Read/Write quorums
  - ⋆ Persistent reliable broadcast
- Read/Write protocols for dynamic server set
- Future work:

  Investigation of feasability conditions