This article appeared in the Proceedings of the 7th IEEE Symposium
on Network Computing and Applications (NCA 2008)

# APART: Low Cost Active Replication for Multi-tier Data Acquisition Systems

Paolo Romano
INESC-ID, Lisbon, Portugal

Diego Rughetti, Francesco Quaglia and Bruno Ciciani
Sapienza Università di Roma

## Abstract

*This paper proposes APART (A Posteriori Active ReplicaTion), a novel active replication protocol specifically tailored for multi-tier data acquisition systems. Unlike existing active replication solutions, APART does not rely on a-priori coordination schemes determining a same schedule of events across all the replicas, but it ensures replicas consistency by means of an a-posteriori reconciliation phase. The latter is triggered only in case the replicated servers externalize their state by producing an output event towards a different tier. On one hand, this allows coping with non-deterministic replicas, unlike existing active replication approaches. On the other hand, it allows attaining striking performance gains in the case of silent replicated servers, which only sporadically, yet unpredictably, produce output events in response to the receipt of a (possibly large) volume of input messages. This is a common scenario in data acquisition systems, where sink processes, which filter and/or correlate incoming sensor data, produce output messages only if some application relevant event is detected. Further, the APART replica reconciliation scheme is extremely lightweight as it exploits the cross-tier communication pattern spontaneously induced by the application logic to avoid explicit replicas coordination messages.*

## 1 Introduction

Recent advances in wireless sensor networks and RFID technology have made sensor driven data acquisition services enter the realms of mainstream applications in a variety of diverse mission critical domains, such as public security, environmental protection, access control and supply chain management. The pervasive diffusion of this class of applications is opening new promising business scenarios and recent economic forecasts anticipate a huge economic growth of the related market in the short term future [21, 22]. However, in order to fully enable the widespread diffusion of mission critical sensor-driven applications it is necessary to develop solutions able to meet their stringent performance and reliability requirements.

From an architectural view point, sensor based data acquisition systems are typically deployed as three-tier systems. The first tier is formed by a set of distributed sensing devices, such as RFID readers, wireless sensor networks or network monitoring probes. The middle-tier servers, also referred to as *sinks*, are in charge to gather, correlate and filter out sensors data, so to propagate only application relevant events towards the back-end tier. The latter tier, in its turn, logs sink originated events, typically by means of transactional components, and exposes these events to user level applications via standard messaging protocols (e.g. WS-RX [17] or XMMP [25]). The system partitioning into multiple tiers provides the potentialities to achieve high modularity and flexibility, reflecting at both software and hardware levels the logical decomposition of applications. On the other hand, ensuring stringent availability and predictability requirements, proper of such mission critical applications, needs replication schemes, to be integrated within the application architectural organization in order to provide timely failure masking mechanisms. Actually, there have been some proposals based on the exploitation of replication in order achieve adequate reliability levels for sensing devices (i.e. the aforementioned first tier) [20]. In this article we focus on the orthogonal issue of exploiting replication at the level of the data acquisition servers, namely the sinks within the middle-tier.

Among the replication approaches developed by the fault-tolerance research community, active replication (also referred to as state machine replication) [24] appears to be a natural candidate to meet the constraints proper of mission critical data acquisition systems. Specifically, with active replication schemes, transparent and instantaneous failover can be achieved by having different replicas of a same server involved in concurrent processing of the incoming messages. This avoids delaying the application output due to failure detection latency, as instead occurs in other replication strategies (e.g. passive replication), thus sensibly enhancing performance predictability in failure-prone environments.

However, in order to ensure server replicas consistency (typically linearizability of the replicated execution history [10]), traditional active replication schemes regulate the evolution of the replicas state trajectories by postponing the processing of incoming messages until the termination of an *Atomic Broadcast* group communication protocol, which is aimed at enforcing replicas agreement on a common, ordered, set of messages to be processed [4]. While a number of optimizations have been proposed to reduce (see, e.g., [6, 18]), or amortize (see, e.g., [8, 15]) the Atomic Broad-

cast cost, this still represents a major source of overhead for active replication.

In this paper, we present APART (A Posteriori Active ReplicaTion), an innovative active replication protocol explicitly tailored for multi-tier data acquisition systems. Unlike traditional active replication schemes, APART does not rely on any replica coordination mechanism (e.g. group communication protocols) to be actuated prior to message processing. It rather enforces replicated sinks consistency only when strictly required, namely when sink replicas externalize their current state by producing an output message. More in detail, the key intuition underlying our proposal is that, since sinks play the role of filters for data provided by sensing devices, they can be abstracted as replicated state machines which only sporadically, yet unpredictably, produce output events in response to the receipt of a (large) set of input messages. In other words sinks can be viewed as state machines of a *silent* type.

APART exploits such a property by avoiding any form of a-priori replica coordination during silent periods, and by relying on an *a-posteriori* reconciliation phase, coordinated by the back-end tier, which is used to correct loss of alignment, if any, of the replicated sinks state trajectories. Also, this is done without generating explicit replica coordination messages. Instead, we exploit the communication pattern spontaneously induced at the application level for event notification towards the back-end tier, ultimately acting as the coordinator for the reconciliation process. Further, the reliance on an a-posteriori replica reconciliation mechanism allows avoiding any restrictive assumption on replica determinism. In other words, the use of APART is not precluded in scenarios with non-deterministic replicas. This makes of APART a highly general solution, which does not require the employment of complex (costly) mechanisms [16, 19] (mandatorily imposed by traditional active replication protocols), explicitly aimed at sheltering the replicas from any source of non-determinism (such as thread scheduling and interrupt handling).

We also carry out a performance evaluation study through a prototype implementation based on Java2 and the Appia communication framework [13]. This study highlights how APART is able to achieve impressive performance levels when compared to state of the art active replication schemes. As it will be shown, even in the case of non-silent replicated servers, APART attains up to 400% throughput increase compared to atomic broadcast, while jointly reducing 4.5 times the latency at low system load. In the case of "moderately" silent replicated servers, the latency reduction with respect to traditional approaches remains on the order of 400%, but the gain by APART in terms of maximum sustainable throughput, inceases up to 1850%.

The remainder of this paper is structured as follows. Section 2 describes the reference system model. Section 3 presents the APART protocol. Related work is discussed in Section 4. The performance study is carried out in Section 5.

# 2 System Model

We consider a classical distributed, asynchronous system model, in which there is no bound on message delay, clock drift or process relative speed. Process communication takes place exclusively through message exchange on top of reliable FIFO communication channels. Hence we assume that each message is eventually delivered, in the same order in which it was originally sent, unless either the sender or the receiver crashes during the transmission [9].

## 2.1 Sensor Processes

As already hinted in the Introduction, there are some literature proposals, based on replication techniques, which are explicitly tailored to reliability at the level of sensors [20]. These can support, e.g., a "good coverage" of the sensed phenomena over time despite failures or other kinds of anomalies. Given that we orthogonally focus on reliability of the data acquisition system, we do not explicitly consider replication techniques for sensing devices in our protocol, and we model the sensing tier as a set of $n$ different sensor processes $\{\text{sensor}_1, \ldots, \text{sensor}_n\}$, each one possibly implemented via the aforementioned reliability oriented techniques. This does not exclude the possibility of crash, and, for simplicity of presentation, we assume that sensor processes do not to recover after a crash. The latter assumption could however be relaxed to admit correct handling of sensors recovery after a crash. This would only imply that sensor processes are able to correctly tag their messages to distinguish messages associated with different incarnations.

Sensor processes provide sinks with a stream of data messages conveying information on sensed environmental phenomena (e.g. temperature values, video/audio samples, RFID tags position, network traffic data for QoS or security purposes etc). We abstract over the details related to the sensing and network management activities, and make no assumptions on the sensor message production rate. We only assume that, according to the active replication paradigm [24], sensors are able to broadcast (via plain best effort broadcast, see, e.g., [9]) their messages to the set of sink processes.

## 2.2 Sink Processes

Logically, the middle-tier is formed by a unique sink. However, we model the sink as a set of $m$ replicated sink processes $\{\text{sink}_1^R, \ldots, \text{sink}_m^R\}$. In other words, replication at the level of middle-tier data acquisition servers is explicitly taken into account in our protocol. Sink processes collect and elaborate the messages arriving from sensor processes. In case the occurrence of any application relevant condition is detected, they generate an output event towards the back-end. Accordingly, locating sink processes in the

proximity of sensor processes can support a reduction of the network traffic associated with the whole application.

The logic hosted by the sink processes can be very diverse and strongly dependent on the specific application domain. However, the production of output events by a sink process is generally triggered either when some statistical metric, computed over the incoming sensors data, reaches predetermined thresholds, or when the sensed data is found to match some known pattern. Also, pattern matching is used by the sink as the basis for triggering output events in a wide class of applications such as, e.g., tracking of RFID tagged objects (whose movement direction is detected by correlating positioning information provided by distributed RFID gates) and distributed intrusion detection systems (which generate alarms in case the data collected by multiple network probes matches some known attack strategies).

We abstract over the detail of the sink application logic and model its behavior through a non-deterministic finite state machine (FSM) [24], whose evolution is determined by invoking the `ProcessMessage` primitive at the sink process. The latter takes a sensor message as input parameter, updates the FSM state and possibly returns an output message encoding the output event destined to the back-end. We use the `null` return value to model the case in which no output message is produced, and say that, in such a case, the FSM is *silent*. In order to quantify the "silentness degree" of the FSM associated with a sink over a given time window we use the parameter $\Sigma = \frac{\#input_msgs}{\#output_msgs}$, where $\#input_msgs$ and $\#output_msgs$ denote the number of invocations of the `ProcessMessage` primitive over the considered time window, and, respectively, the number of times this primitive does not return `null`. It is straightforward to see that feasible values for $\Sigma$ lie in the interval [1,+$\infty$), where the left-end value, namely $\Sigma = 1$, corresponds to the case of FSMs producing an output event for each input message, and the right-end value, namely $\Sigma = +\infty$, is representative of scenarios where no output event is ever produced despite the arrival of whichever amount of input messages. Obviously, the aforementioned filtering and network traffic reduction effects will depend (beyond sink processes locations) also on the actual value of $\Sigma$.

We additionally assume that two other primitives are available at the sink process. namely `getFSMState` and `setFSMState`. The former primitive returns the current state of the FSM associated with the sink process, while the latter primitive replaces the current sink process state with the one passed as input parameter (in other words it reinstalls the state of the sink process).

Finally, we assume that sink processes do not recover after a crash. However, we assume that at least one sink process in the set $\{\text{sink}_1^R, \ldots, \text{sink}_m^R\}$ is correct, i.e., it does not crash. Hence, we tolerate the crash of at most $f < m$ sink replicas. These assumptions are introduced to simplify the protocol presentation, and could be relaxed by relying on some orthogonal group membership service [1] aimed at notifying the changes in the group of replicated sink processes (caused by both crash and recover events).

### 2.3 Back-end Data Server

The system back-end consists of a data server process which receives the sinks output events and registers them within a local database, used for event publication towards external applications. We do not explicitly model the mechanisms used to publish the events, as these are orthogonal to the APART protocol, and abstract over the details of database updates via a `PublishTransaction` primitive. The latter takes two input parameters, namely a unique identifier and a sink output message (representative of the event to be published), and executes the transactional logic that inserts such a tuple within the database. We assume the execution of the `PublishTransaction` primitive with a given input identifier to be idempotent, i.e. no two transactions associated with the same input identifier can ever be committed. This can be achieved by storing the identifier passed as input parameter in a dedicated user level database table within the boundaries of the application level transaction and leveraging standard primary key integrity constraints to filter out duplicate transactions [23].

Additionally, we assume that the data server has access to a log on stable storage, which preserves its state in the APART protocol despite crashes. Stable storage is used to persist a single tuple conveying information on the protocol current state. At this end, we assume the presence of the primitive `log`, which records the tuple passed as input parameter onto stable storage, and of the primitive `readFromLog`, which simply returns the value of the currently logged tuple, or the value `null` in case no tuple has been yet logged.

We assume that the back-end data server eventually recovers after a crash and that there is a time after which it stops crashing and remains up, allowing outgoing messages to be eventually delivered to all the correct sink replicas. In practice, this means assuming that the data server can experience a period of instability during which it can crash and recover, and then a period during which it does not crash, which is long enough to allow the conclusion of an interaction round with correct sink processes.

## 3 The APART Protocol

In this section we first present an overview of the APART protocol and then formalize it by providing the pseudo-code description of the sink and the back-end data server processes. We omit detailing the pseudo-code of the sensor processes as they simply broadcast SENSORMSG messages to the set of sink replicas, along with the following information: $msgId$, namely a sequentially increasing identifier,

and $data$, which conveys information related to the sensed phenomenon.

## 3.1 Overview

Unlike classical active replication schemes, in APART sink replicas do not run any coordination protocol (such as, e.g., atomic broadcast [4] or consensus [7]) to ensure an "a-priori" agreement on the ordered set of sensor messages to be processed. Conversely, sink replicas process incoming sensor messages as soon as they are received and rely on an "a-posteriori" coordination phase. The latter is triggered when the sink FSM produces an output event (thus saving any coordination overhead during silent periods) and enforces the coherency of all the sink replicas states before these start processing any additional sensor message.

The information exploited in the a-posteriori coordination phase includes not only the state of the local FSM associated with sink replicas, but also the state of the communication channels towards the sensors, which is concisely encoded by a vector clock [12]. The latter information is required to ensure that any replica $\text{sink}_i^R$ which, according to the outcome of the a-posteriori coordination phase, had to reinstall its state to the state value associated with a different replica $\text{sink}_j^R$, is able to perform the following tasks:

1. Determine if it has already processed some sensor message not yet received/processed by $\text{sink}_j^R$. These messages must in fact be reprocessed by $\text{sink}_i^R$ after a-posteriori coordination, in order to to ensure at-least-once semantic for the processing of each message produced by a sensor process that does not eventually crash ([1]). This guarantees that messages from correct sources are not eventually excluded along the reconciliated FSM state trajectory, thus preventing biased observations of the sensed phenomena caused by the exclusion of valid data. To enable message reprocessing after a state reinstall operation, sink processes maintain the received sensor messages in a volatile buffer. This is pruned out of any obsolete message (i.e. messages known to be already processed by $\text{sink}_j^R$ along the trajectory representative of reconciliation) at each coordination round.

2. Detect if $\text{sink}_j^R$ has already processed some message not yet received by $\text{sink}_i^R$. These messages must be discarded by $\text{sink}_i^R$ (if eventually received), since they have been already incorporated into the reconciliated FSM trajectory. This must be done in order to ensure at-most-once processing of sensor messages.

In APART the a-posteriori coordination phase is mastered by the back-end data server, which waits for $minProposals$ output events from the replicated sink processes, selects (and accepts) one of them, and then broadcasts it back to all the sink replicas. Note that the choice to rely on the back-end data server to coordinate sink replicas allows merging into a single phase both replica consistency management, and the externalization of output events towards back-end applications.

Concerning $minProposals$, as well as the logic driving the selection of the representative sink output event (among the proposals), over which we abstract by means of the `select` primitive, these are treated as tunable protocol parameters. Legal values for $minProposals$, matching failure resiliency assumption for replicated sink processes, span in the domain [1,m-f]. Also, the real selected value allows trading-off the latency of output production vs the data server ability to filter out "anomalous" output events. In fact, given that any divergence in the states of the replicated sink processes is corrected by forcing all replicas to coherently reinstall a same state ([2]), it is guaranteed that any output event produced by a sink replica is representative of a linearizable processing history [10]. Hence, since all the output events generated by sink replicas provide linearizability semantic, the back-end data server could just set $minProposals = 1$, with the objective to select the first output event it receives as the representative event, thus enabling timely event delivery, and timely activation of a-posteriori replica coordination. On the other hand, by setting $minProposals > 1$, the back-end data server can actuate some voting procedure to select a specific processing history linearization among those externalized by the sink replicas through their output events. At this end, the data server might rely either on general-purpose logics, such as majority voting strategies, or on more complex ones, leveraging the application semantic, such as discarding output events generated by sinks whose vector clocks highlight missed message receipts from a specific subset of sensor processes.

## 3.2 Sink Behavior

The pseudo-code for sink processes is shown in Figure 1. A sink maintains (i) a *msgBuffer* used to buffer incoming messages, which is assumed to provide FIFO semantic, (ii) a vector clock *localVC*, keeping track of communication histories with sensor processes, and (iii) a sequentially increasing identifier, *curRoundId*, which is used to tag output events destined to the back-end. Further, the sink relies on the *awaitingDecision* boolean variable to temporarily inhibit the processing of incoming messages in case it is waiting for a reply (i.e. for a decision) from the back-end data server for a previously externalized output event.

If the sink process receives a message from a sensor process while it is not waiting for a decision (i.e. *awaitingDecision* is set to false), the local vector clock is used

[1] Recall that a message broadcasted by a correct sensor process is eventually received by any correct sink replica.

[2] Divergencies are imputable to, e.g, different processing orders of the incoming sensor messages, or to non-determinism of the sink processes FSMs.

```
FIFOQueue msgBuffer;              //FIFO ordered message buffer
VectorClock localVC[n];           //sensor messages history
int curRoundId=0;
boolean awaitingDecision=false;

upon receive(SENSORMSG, msgId, data) from sensor_i ∧ ¬awaitingDecision do
  if (msgId > localVC[sensor_i])  //filter out obsolete sensor messages
    msgBuffer.push([msgId,sensor_i,data]);  //buffer the incoming message
    handleInputMsg(msgId, sensor_i, data);

upon receive(DECISION, roundId, FSMState, vectorClock) from back-end server do
  if (roundId ≥ curRoundId)       //filter out obsolete coordination messages
    unsetRetransmissionTimeout(); //unset the retransmission timeout
    curRoundId = roundId;         // set round counter
    setFSMState(FSMState);        //set FSM internal state
    localVC = vectorClock;        //update local vector clock
    clearBuffer();                // prune message buffer
    awaitingDecision=false;       //unblock processing of sensor messages
    while (msgBuffer ≠ ∅ ∧ ¬awaitingDecision) do
      [msgId,sensor_i,data]=msgBuffer.pop();
      handleInputMsg(msgId,sensor_i,data);  //reprocess buffered messages

upon timeoutExpired do     //triggers retransmission of the latest output event
  send(OUTPUTEVENT, courRoundId, outputMsg, getFSMState(), localVC)
                to back-end server;
  setRetransmissionTimeout();     //set the retransmission timeout

void handleInputMsg(int msgId, int source, SensorData data)
  localVC[source] = msgid;        //update the corresponding vector clock entry
  Message outputMsg=ProcessMessage(data);   //FSM update
  if (outputMsg ≠ null)           //the FSM produces an output event
    curRoundId++;                 // update output messages counter
    send(OUTPUTEVENT, curRoundId, outputMsg, getFSMState(), localVC)
                to back-end server;
    setRetransmissionTimeout();   //set the rentransmission timeout
    awaitingDecision=true;        // block processing of sensor messages

void clearBuffer()                // message buffer pruning
  ∀msg ∈ msgBuffer where msg.id ≤ localVC[msg.source]
    msgBuffer.remove(msg);
```

**Figure 1. Sink Process Behavior.**

to detect whether the sensor message has already been incorporated into the execution history currently seen by the sink ([3]). In the positive case the message is simply discarded. Otherwise the sink buffers the message, updates its vector clock to reflect the message receipt and invokes the `ProcessMessage` primitive to feed its FSM with the sensor data. If the FSM is silent, namely `ProcessMessage` returns `null`, the sink starts waiting again for incoming messages. Otherwise, it delivers the FSM output event to the back-end data server by means of an OUTPUTEVENT message, piggy-backing the current output event identifier, namely *curRoundId*, the state of the local FSM, retrieved via the primitive `getFSMState`, and the local vector clock. Also, in order to ensure the termination of the coordination phase despite crashes of the back-end data server, the sink periodically re-transmits the OUTPUTEVENT message towards the back-end. Finally, the sink sets *awaitingDecision* to true so to temporarily suspend the processing of any additional incoming sensor message.

[3]Recall that this may happen in case the sink has installed the state of a different replica that already received and processed that message.

```
set sinkSet = {sink_1,...,sink_m};
Set proposals = {};
int curRoundId = 1;

upon receive(OUTPUTEVENT, roundId, outputMsg, FSMState, localVC)
          from sink_i ∧ curRoundId=roundId
  proposals.add([outputMsg,FSMState,localVC]);
  if | proposals | ≥ minProposals do
    [outputMsg,FSMState,VC]= select(proposals);
    log([curRoundId,outputMsg,FSMState,VC]);
    send(DECISION,roundId,FSMState,VC) to each sink_i ∈ sinkSet;
    PublishTransaction(curRoundId,outputMsg);
    curRoundId++;
    proposals={};

upon recoverFromCrash do
  if (([roundId,outputMsg,FSMState,VC] = readFromLog()) ≠ null)
    send(DECISION,roundId,FSMState,VC) to each sink_i ∈ sinkSet;
    PublishTransaction(roundId,outputMsg);
    curRoundId=roundId+1;
```

**Figure 2. Back-end Data Server Behavior.**

If a DECISION message is received from the back-end server, the sink first makes sure that this is not an obsolete message (associated with some obsolete coordination phase). Then it unsets the timeout used to retransmit the OUTPUTEVENT message and updates the local FSM state, its local vector clock and *curRoundId* according to the payload of the received DECISION message. Next it unsets *awaitingDecision* to signal the termination of the coordination phase, and removes any obsolete message from its buffer, which results as already incorporated in the execution history of the sink replica whose state has been selected as representative for replicas reconciliation (i.e. whose state is installed by all the other replicas as the effect of a-posteriori coordination). This is done on the basis of the vector clock piggybacked on the DECISION message. Finally, it starts reprocessing sensor messages left in its buffer.

### 3.3 Back-end Data Server Behavior

Figure 2 shows the back-end data sever pseudocode. The main data structures kept by the data server are (i) a monotonically increasing identifier, namely *curRoundId*, which is used to keep track of the current round of interaction with sink processes, and (ii) a set, namely $proposals$, used to gather the OUTPUTEVENT messages received from the sinks in the current round.

If an OUTPUTEVENT message associated with the current round is received, this is inserted in the $proposals$ set. As soon as the cardinality of the $proposals$ set reaches the value $minProposals$, the data server invokes the `select` primitive to choose the output event selected as representative for publication (as well as the corresponding sink replica state and vector clock), logs the choice on persistent storage and sends back the decision (i.e. the result of the selection) to the sink replicas. Then it invokes the `PublishTransaction` primitive passing the current

round identifier and the selected output event as input parameters, so to execute the corresponding database update. Finally, it empties the $proposals$ set and increments the round counter.

Upon recovery after a crash, the data server retrieves from the log the information related to its latest decision, sends back a DECISION message to the sink replicas and invokes the `PublishTransaction` primitive. Note that, being these operations idempotent, they can be safely executed multiple times (e.g. in the case of multiple subsequent crashes of the data server). Finally, the current round identifier is updated on the basis of the one retrieved from the log.

## 4 Related Work

Replication solutions are typically classified as either passive (also known as primary-backup) [2] or active [24].

In active replication, the coherency of the replicas, which are assumed to be deterministic, is guaranteed by delaying the processing of incoming messages until the termination of an Atomic Broadcast (AB) [4] protocol aimed at ensuring the agreement on a common message processing order. The works in [6, 18] try to reduce the AB overhead by employing optimistic approaches based on the idea to exploit the *spontaneous ordering* property over LANs so to overlap AB with message processing. Other works, such as [14], have introduced mechanisms aimed at "inducing" spontaneous ordering also over WANs [14], so to reduce the abort probability of optimistic-AB protocols. Another orthogonal technique for reducing the AB overhead [8, 15] is based on delaying the activation of the AB protocol, so to order a batch of input messages with a single run. This approach has been shown to provide remarkable performance benefits at high system load, where replicas more likely have to order multiple input messages over short time windows. The semi-active replication scheme proposed in [19] allows relaxing the assumption on replica determinism. Each time replicas have to execute a non-deterministic step, they rely on a process, called the leader, to make and notify the choice to be followed by all the replicas.

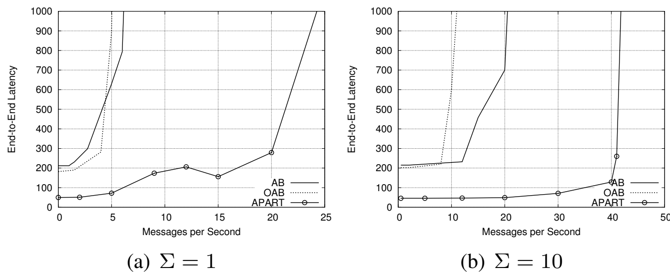
The aforementioned active replication schemes rely on *a-priori* replica synchronization, which ensure that any state update occurs only after having ensured its durability, as well as the replicas agreement on message delivery order (i.e. if a replica processes an update in some order, then any correct replica eventually processes it in the same order). Conversely, APART relies on *a-posteriori* synchronization, which is triggered only if replicas externalize their state, and which relies on the back-end data server to enforce state durability and consistency across the replicated sinks (i.e. if the data server logs a state update, then every correct sink eventually aligns its state trajectory to the logged one).

In passive replication schemes a single process, referred to as the primary, is in charge of processing incoming requests and of delivering state updates to its backup replicas. Like APART, passive replication can handle non-determinism. However, given that there is no primary process in APART, our proposal avoids system transient unavailability that is caused, in the event of a primary failure, by a reconfiguration procedure aimed at electing a new primary (recall that this also entails the latency for detecting the failure of the original primary). Recently, a variant of the classical passive scheme, named semi-passive replication, was proposed in [5], which does not rely on a fixed primary, but is rather based on the rotating coordinator paradigm [3]. It allows using more aggressive failure detection timeouts without incurring the cost for updating group membership [1], which helps reducing the fail-over latency. However, there is no complete masking of failure occurrences as in APART, and more in general in active replication strategies.

## 5 Performance Evaluation

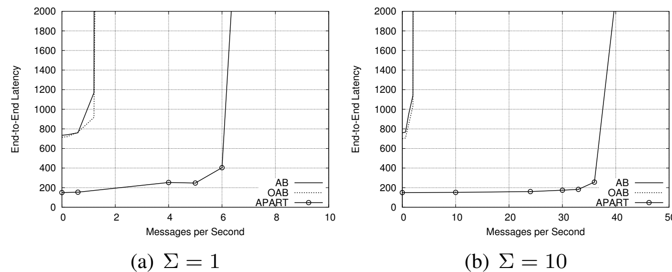
This section is devoted to compare the APART protocol with classical active replication techniques. To carry out the evaluation, we developed a prototype implementation of a multi-tier middleware for data acquisition, which is based on the Appia group communication framework [13]. Our study is focused on system normal behavior, namely the scenario in which no failure suspicion occurs. In the comparison, we consider a sequencer based uniform total order algorithm [9], which we will refer to as *AB* in the following. This choice is motivated by the fact that this protocol achieves the lower bound for Atomic Broadcast message delivery latency [11]. We compare the performance of the APART protocol also with the one provided by the optimistic atomic broadcast protocol in [18], hereafter named *OAB*. This protocol delivers broadcast message as soon as they are received from the network, thus immediately activating their processing, but waits for the termination of a sequencer based uniform total order algorithm, prior to processing any further incoming message or producing any output event. In other words, there is an overlapping between the processing phase and the execution of the atomic broadcast validating the processing order. This is effective in case the incoming messages are spontaneously received in the same order by all the replicas. On the other hand, OAB requires replica rollback if the spontaneous message delivery is found to differ from that determined by the uniform total order algorithm.

To the best of our knowledge, there is currently no standard benchmark for data acquisition systems. Hence, in our performance evaluation study we choose to rely on a synthetic application. We model the processing of an incoming sensor message at sinks by introducing 20 milliseconds delay (implemented as a non-costly sleep operation). In order to determine whether the processing of a sensor message should trigger the generation of an output event we rely on




(a) $\Sigma = 1$ (b) $\Sigma = 10$

**Figure 3. Average Latency (LAN Scenario).**




(a) $\Sigma = 1$ (b) $\Sigma = 10$

**Figure 4. Average Latency (WAN Scenario).**

$\Sigma$ (see Section 2), which we treat as an independent parameter in the study, and for which two different values have been adopted, namely 1 and 10. The former value models a worst case scenario for APART, in which FSMs are non-silent. On the other hand, the latter value allows us to quantify the performance benefits achievable by the APART protocol while modeling "moderately silent" FSMs.

In order to evaluate different deployments, we artificially delay the inter-process communication (which is layered on top of TCP channels). Specifically we consider two opposite scenarios, representative of local vs geographical scale distribution of system components. The LAN scenario is emulated by delaying messages by a time interval exponentially distributed in the range [1-10] milliseconds, with mean value equal to 2 milliseconds. For the WAN scenario, the message delay latencies were determined according to an exponential distribution constrained in the range [50-100], with mean value equal to 75 milliseconds.

Our test-bed consists of two machines, namely an AMD Sempron 2400+ with 1GB of RAM, and an Intel Core 2 Duo T7250 with 2GB of RAM, both running Linux (Kernel 2.6.22-14). The former machine hosts six sink replicas, whereas the latter one is used both to generate sensor messages (via 12 RFID reader emulators) and to host the data server logic. Note that the choice to deploy the sensors and the data server on the same physical machine allows us to accurately measure the end-to-end message latency for traversing the whole chain of components (i.e. sinks and data center, as well as communication channels), without the need for using (intrusive) clock synchronization protocols. In Figures 3 and 4 we report the average end-to-end message latency for LAN and WAN scenarios, respectively. This is evaluated as the time interval between the production of a sensor message and the publication of the corresponding output event at the back-end data server via the `PublishTransaction` primitive.

Figure 3(a) shows the performance of the compared protocols in the case of non-silent sink replicas ($\Sigma = 1$), highlighting how, even in this adverse scenario, the APART protocol sensibly outperforms state of the art active replication schemes both at low and high system load. In fact, at low loads, e.g. around 1 msgs/sec, the average end-to-end latency for APART is on the order of 50 milliseconds, whereas both AB and the OAB are nearly 4 times slower, achieving, respectively, 210 milliseconds and 190 milliseconds processing time. For what concerns the maximum throughput, APART reaches the saturation point at around 20 msgs/sec, whereas AB and OAB saturate at around 5 msgs/sec. Such a striking performance gain for the APART protocol, even in this worst case scenario, is mainly imputable to the avoidance of message handling costs proper of atomic broadcast, which provides benefits especially when the message traffic increases.

The plots in Figure 3(b) show how the APART performances can sensibly improve as $\Sigma$, namely the replica silentness degree, increases. With respect to the case of a non-silent FSM, here the maximum throughput attained by APART doubles, overcoming 40 messages per seconds and falling very close to the theoretical throughput achievable by the system, namely 50 messages per second (recall that the processing of each message by a sink replica takes 20 milliseconds). Overall, the maximum throughput of APART is twice that of AB, and four times that of OAB which starts trashing much earlier than AB due to high rollback frequency as the load increases.

The performance gap between APART and the two considered atomic broadcast protocols drastically increases when considering the WAN scenario, as shown in Figure 4. In this case, in fact, the increase in the message transmission latency translates into a sensible increase of the atomic broadcast latency, which becomes the main system bottleneck for both the AB and OAB protocols limiting their maximum throughput to around 1.5 msgs/sec, independently of the considered value of $\Sigma$. The increased message transmission latency has also an impact on the performance of the APART protocol in the worst case scenario of non-silent FSM, see Figure 4(a), which drops from the 20 msgs/sec of the corresponding LAN scenario, to around 6 msgs/sec. This is due to that, in the WAN scenario, the bottleneck for the APART protocol is represented by the coordination phase driven by the data server. Since this takes two communication steps, its average latency, according to our system settings, is of about 150 milliseconds. This places an upper bound on the maximum throughput achiev-

able by APART in this specific configuration at about 6.7 msgs/sec. On the other hand, as $\Sigma$ is increased to 10, see Figure 4(b), the coordination phase does not result any more to be the system bottleneck, as this is triggered only after having processed, in average, 10 sensor messages. This justifies the sixfold relative increase of the maximum sustainable throughput (with respect to the case of non-silent FSMs) which in this configuration is around 37 msgs/sec for APART. Finally, it is noteworthy to highlight that, in the WAN scenario, APART reduces the low load end-to-end latency with respect to AB and OAB by a factor of 4.5 independently of the considered $\Sigma$ value, and increases the maximum sustainable throughput by a factor of 4, when $\Sigma$=1, and by a factor of 18.5, when $\Sigma$=10.

## 6 Conclusion

In this paper we presented APART, a novel active replication protocol specifically tailored for multi-tier data acquisition systems. Unlike traditional active replication schemes, APART does not rely on an a-priori replica coordination mechanism, such as atomic broadcast, to guarantee the replicas agreement on a common total order of the execution history. Conversely, APART enforces replica consistency a-posteriori (i.e. only when sink replicas externalize their current state by producing an output message), exploiting the cross-tier communication patterns spontaneously induced by the application logic to avoid generating explicit replica coordination messages.

The reliance on an a-posteriori, rather than a-priori, replica coordination schemes allows APART to manage non-deterministic replicas with no additional overhead or complexity. While relaxing this base assumption of classical active replication solutions has clearly a theoretical relevance, it is worthy highlighting that it has also a strong pragmatical impact. The need to enforce replica determinism in real life systems, in fact, often requires the employment of costly error prone mechanisms, e.g. [16, 19] aimed at sheltering the replicas from any source of non-determinism.

Further, APART design choice to rely on an a-posteriori replica coordination, allows it to strikingly outperform state of art active replication schemes, especially in the case of *silent* replicated servers, namely servers that only sporadically, yet unpredictably, produce output events in response to the receipt of a (possibly large) volume of input messages. This is a common scenario in data acquisition systems, where processes act as sinks that filter and/or correlate incoming sensor data.

## References


[1] K. P. Birman. *Distributed Computing with the Isis toolkit*. IEEE Computer Society Press, 1994.

[2] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *The primary-backup approaches*. ACM Press/Addison-Wesley Publishing Co., 1993.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[4] D. Powell (ed.). *Special Issue on Group Communication*, volume 39. ACM, 1996.

[5] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, SwitzerlandMa, 2000.

[6] P. Felber and A. Schiper. Optimistic active replication. *Proc. of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 333–341, Washington, DC, USA, 2001. IEEE Computer Society.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[8] R. Friedman and E. Hadad. Adaptive batching for replicated servers. *Proc. of the 25th Symposium on Reliable Distributed Systems (SRDS)*, pages 311–320. IEEE Computer Society, 2006.

[9] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[10] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. *SIGACT News*, 32(2):45–63, 2001.

[12] F. Mattern. Virtual time and global states of distributed systems. *Proc. of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[13] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. *Proc. of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 707–710. IEEE Computer Society, 2001.

[14] J. Mocito, A. Respicio, and L. Rodrigues. On statistically estimated optimistic delivery in large-scale total order protocols. *Proc. of the 12th IEEE International Symposium on Pacific Rim Dependable Computing (PRDC)*, pages 202–209. IEEE Computer Society, 2006.

[15] P. Narasimhan, L. Moser, and P. Melliar-Smith. Message packing as a performance enhancement strategy with application to the Totem protocols. *Proc. of the 39th Global Telecommunications Conference (GLOBECOM)*, pages 649–653 vol.1, 1996.

[16] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. *Proc. of the 18th Symposium on Reliable Distributed Systems (SRDS)*, pages 263–273. IEEE Computer Society Press, 1999.

[17] OASIS. Web Services Reliable Messaging, 2008.

[18] F. Pedone and A. Schiper. Optimistic atomic broadcast. *Proc. of the 12th International Symposium on Distributed Computing (DISC)*, pages 318–332. Springer-Verlag, 1998.

[19] D. Powell, M. Chérèque, and D. Drackley. Fault-tolerance in Delta-4*. *ACM Operating Systems Review (SIGOPS)*, 25(2):122–125, 1991.

[20] A. Rahmati, L. Zhong, M. Hiltunen, and R. Jana. Reliability techniques for rfid-based object tracking applications. *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 113–118, Edinburgh, UK, 2007. IEEE Computer Society.

[21] Research and Market. RFID industry a market update, July 2005.

[22] RNCOS. Global RFID market analysis till 2010, December 2007.

[23] P. Romano, F. Quaglia, and B. Ciciani. A lightweight and scalable e-Transaction protocol for three-tier systems with centralized back-end database. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1578–1583, 2005.

[24] F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[25] XMPP Standards Foundation (XSF). Extensible Messaging and Presence Protocol (XMPP) 1.0, 2003.