APART$^+$: Boosting APART Performance via Optimistic Pipelining of Output Events

Paolo Romano
INESC-ID, Lisbon, Portugal

Francesco Quaglia and Bruno Ciciani
Sapienza Università di Roma, Italy

Abstract

APART (A Posteriori Active ReplicaTion) is a recently proposed active replication protocol specifically tailored for multi-tier data acquisition systems. It ensures consistency of middle-tier sink replicas by means of an a-posteriori synchronization phase based on reconciliation, which is activated only in case replicas react to an input message from the sensors by generating an output event destined to the back-end tier.

This paper enhances APART via a novel non-blocking synchronization scheme which prevents replicas from stalling while waiting for the outcome of an on-going synchronization phase. Contrarily, replicas are allowed to optimistically process data from the sensors, and to immediately propagate any output event towards the back-end tier. The removal of the blocking synchronization phase from the critical path gives rise to striking performance gains via an effective overlapping of event processing and synchronization. On the other hand, system consistency is ensured by enhancing the back-end tier synchronization logic in order to filter out optimistically produced output events that are incompatible with the reconciled state trajectory.

1 Introduction

Over the years we have witnessed the pervasive adoption of sensor driven data acquisition systems in a variety of mission-critical application domains, such as public security, environmental protection, access control and supply chain management. From an architectural perspective, such systems can be viewed as three-tier systems. The sensing devices, such as RFID readers, sensor networks or network monitoring probes, compose the first tier. The data streams produced by these devices are gathered, correlated and filtered by middle-tier servers, also referred to as *sinks*, whose role is to identify relevant events and to propagate them towards the back-end tier. The latter tier is in charge of archiving the output events generated by the sinks and of making them available to user level applications via standard interfaces (e.g. WS-RX [12]).

One of the main challenges in these safety-critical systems is how to guarantee that the incoming data streams are processed under strict timing constraints, while jointly providing strong dependability guarantees in terms of high availability and failure resiliency [14]. Clearly, in order to ensure adequate fault-tolerance levels, proper mechanisms need to be employed across all the tiers (i.e., sensors, sinks and back-end) of the data acquisition system. In this paper we focus on the issue of how to efficiently replicate middle-tier sinks for fault-tolerance purposes. Orthogonal replication solutions aimed at enhancing the reliability of the sensing devices and the back-end tier can be found, respectively, in, e.g., [16] and [8].

Among the replication schemes presented in literature, active replication (AR) [20] appears to be a natural candidate to meet the constraints of mission critical data acquisition systems. In fact, AR ensures transparent and instantaneous fail-over, providing good performance and latency predictability even in failure-prone environments. The common idea underlying the numerous variants of AR presented in literature [2] is to first reach an agreement among the replicas [4] on a common processing order for the incoming messages, which are then actually processed in parallel. Such an a-priori agreement, together with the assumption on replicas determinism, suffice to ensure strong replicas consistency. On the other hand, enforcing agreement on the processing order of each incoming message introduces a remarkable overhead which can significantly impact performance [2, 4, 19].

In a recent paper [19], we have introduced APART (A Posteriori Active ReplicaTion), an innovative AR scheme that, by leveraging some key features of multi-tier data acquisition systems, provides two main benefits with respect to state of the art AR solutions: (i) the removal of replicas determinism assumptions, which very often obliges AR schemes to rely on additional, complex mechanisms aimed at filtering out any source of non-determinism (such as thread scheduling [10], or the interaction with external devices); and (ii) a strong reduction of the replica synchronization overhead, which allows APART to achieve remarkable performance gains in terms of both maximum sustainable throughput, and latency reduction at low system load.

The main intuition underlying the APART protocol is that, in modern multi-tier data acquisition systems, sink components are not only in charge of processing the incoming data streams, but also play the role of filters for the data generated by the sensing devices. In other words, sinks can be abstracted as *silent* state machines which only sporadically, yet unpredictably, produce output events in re-

sponse to the receipt of a (possibly large) set of input messages. APART exploits such a property by avoiding any form of replica coordination in silent periods (i.e. periods during which incoming data are processed without generating any output event), and by triggering an *a-posteriori* replica reconciliation phase *only* when a sink externalizes its state by generating an output event. This is done without using explicit replica coordination messages, but rather by exploiting the communication pattern spontaneously induced at the application level for event notification towards the back-end tier. The latter acts as the coordinator of the replica synchronization process, which aims at installing the reconciled state value on all the diverging replicas, if any.

However, the APART's reconciliation scheme operates in a blocking mode, thus forcing all sink replicas to temporarily suspend processing sensor messages. This prevents replicas with inconsistent states from generating output events, and suffices to ensure that any output event received by the back-end tier is associated with an admissible state trajectory. On the other hand, the presence of a blocking coordination scheme on the critical path of the end-to-end interaction directly translates into an upper bound on the system's throughput (expressed in terms of number of output events accepted by the back-end tier per time unit).

In this paper we present APART$^+$, an enhanced version of the APART protocol which relies on a novel reconciliation scheme allowing sink replicas to keep on processing incoming sensor data (and to possibly propagate any associated output event) in an optimistic fashion, without waiting for the completion of any previously activated reconciliation phase. On one hand, this implies the need to deterministically discard the output events optimistically generated by replicas in an inconsistent state. On the other hand, this allows the set of sink replicas whose state is eventually chosen by the reconciliation phase to seamlessly and uninterruptedly process incoming data. Such replicas, de facto, do not incur any stall of their processing activities caused by the a-posteriori synchronization scheme.

We quantify the performance benefits from APART$^+$ through a simulation study, which highlights the striking gains in the maximum achievable throughput with respect to APART, which extends up to a 20x increase for data acquisition systems deployed over wide area, or relatively slow, networks.

The rest 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

The system model considered in this paper is the same as the one in [19], and is here recalled for self-containment. We consider a classical distributed, asynchronous system model, in which there is no bound on message delay, clock drift or process relative speed. Communication takes place exclusively through message exchange on top of reliable FIFO communication channels, i.e. every message is eventually delivered, in the same order in which it was originally sent, unless either the sender or the receiver crashes [4] during the transmission.

2.1 Sensor Processes

We model the sensing tier as a set of n distinct sensor processes $\{\text{sensor}_1, \ldots, \text{sensor}_n\}$. Note that we do not explicitly consider replication of sensing devices, even though redundancy techniques at the sensors level [16] could be leveraged in order to ensure adequate sensing accuracy. On the other hand, we do not exclude the possibility of crash, and, for simplicity of presentation, we assume that sensor processes do not to recover after a crash.

A sensor process generates a stream of 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.) to the sinks. We abstract over the details related to the sensing activities, and only assume that, according to the active replication paradigm [20], sensors are able to broadcast (via plain best effort broadcast, see, e.g., [4]) their messages to the set of sink processes.

2.2 Sink Processes

We assume the existence of a set of m replicated sink processes $\{\text{sink}_1^R, \ldots, \text{sink}_m^R\}$, providing the illusion of a single, highly-available sink. Sink replicas process the messages arriving from the sensors, and generate output events towards the back-end tier if the occurrence of any application relevant condition is detected. The logic hosted by sink processes is dependent on the specific application domain. In general, the production of output events by a sink process is triggered either when some statistical metric, computed over the incoming sensors data, reaches predetermined thresholds, or when the data is found to match some known pattern.

We abstract over the details of the sink application logic and model its behavior through a non-deterministic finite state machine (FSM) [20], whose evolution is determined by invoking the `ProcessMessage` primitive. This primitive takes a sensor message as input parameter, updates the FSM state and possibly returns an output event destined to the back-end. We use the `null` return value to model the case in which no output event 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`.

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 state of the sink process 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.

2.3 Back-end Data Server

The system back-end consists of a data server process which receives output events from the sinks and registers them within a local database, used to make events available to external applications. We do not explicitly model the mechanisms used to publish the events, which 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, see, e.g., [18, 17]. Additionally, we assume that the data server has access to a log on stable storage, which preserves its current state in the APART$^+$ protocol by persisting a single tuple. 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.

The back-end data server is assumed to eventually recover after a crash. It is further assumed that there is a time after which the back-end data server 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 order to better highlight the differences between APART$^+$ and the original APART protocol, we first recall the main structure and features of APART. Then, we provide an overview of the novel features of APART$^+$, together with the pseudo-code formalization for the behavior of sink and back-end data server processes. We omit detailing the pseudo-code for 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 of the APART Protocol

As hinted, in APART sink replicas do not run a coordination protocol (such as, e.g., atomic broadcast [2] or consensus [4]) to ensure an "a-priori" agreement on the processing order of the incoming sensor messages. Conversely, sink replicas rely on an "a-posteriori" coordination phase, which is triggered whenever the sink FSM produces an output event (thus saving any coordination overhead in silent periods). During the coordination phase, $sink_i^R$ stops processing incoming data from the sensors, awaiting for the back-end data server to decide which one, among the (possibly diverging) output events/state trajectories generated by the sink replicas, should be globally accepted by the whole replicas' set. If the coordination phase decides to accept replica $sink_j^R$'s (where $j \neq i$) output event/state trajectory, and this differs from that generated by $sink_i^R$, then $sink_i^R$ is forced to install the local state of $sink_j^R$ before processing further incoming messages.

In the a-posteriori coordination scheme, the sink piggybacks two main pieces of information on its output event message: (i) the state of the local FSM, and (ii) the state of the communication channels towards the sensors, concisely encoded by a vector clock [9]. The latter information is required to ensure that any replica 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 sink_j^R, is able to perform the following tasks:

1) Determine if it has already processed some sensor messages not yet received/processed by sink_j^R. These messages must in fact be reprocessed by sink_i^R after the a-posteriori coordination, in order to to ensure at-least-once processing semantic. 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 sink_j^R along the trajectory representative of reconciliation) at each a-posteriori coordination round.

2) Detect if sink_j^R has already processed some message not yet received by sink_i^R. These messages, which have been already incorporated into the reconciled FSM trajectory, must be discarded by sink_i^R to ensure at-most-once processing semantic.

In the a-posteriori synchronization phase, the back-end data server waits for $minProposals$ output events from the replicated sink processes. Afterwards, the data server selects (and accepts) one of them, and broadcasts it back to all the sink replicas. The value of $minProposals$, as well as the logic driving the selection of the sink output event, are treated as tunable protocol parameters, which allow trading-off the latency of output production vs the data server ability to filter out "anomalous" output events. In fact, since APART guarantees that any output event produced by a sink replica is representative of a linearizable processing history [5], the back-end data server could just set $minProposals = 1$, with the objective to externalize

the output event as soon as possible, i.e. as soon as the first output event from whichever sink is received. On the other hand, by choosing larger values of $minProposals$ in the admissible range, [1,m-f], the back-end data server could leverage some voting scheme to select a specific processing history linearization among those externalized by the sink replicas.

3.2 From APART to APART$^+$

Unlike APART, where sink replicas interrupt the message processing activity while waiting for the a-posteriori coordination phase to be concluded, APART$^+$ allows sink replicas to continue processing the data incoming from the sensors and to generate new output events, even if there are on-going coordination phases. This is achieved by relying on more refined state management techniques which allow to identify, and deterministically discard, any output event that is *not* representative of the eventually selected linearizable processing history.

In order to allow the back-end data server to detect any non-linearizable (optimistic) output event, the sink externalizes output events towards the back-end via two different types of messages, depending on whether the event is generated optimistically (i.e. based on the processing of sensor messages occurred while there are still pending coordination phases) or not. If the event is generated in a conservative manner, like in APART the output message piggybacks the local FSM state and the corresponding vector clock [9]. On the other hand, if the output event is generated optimistically, the sink output message also piggybacks the digest (computed through a standard cryptographic hash function, such as MD5 or SHA1 [11]) of its FSM state at the time of the generation of the previous output event. This information is exploited by the back-end data server during the coordination phase in order to identify and filter out any optimistic output produced by sink replicas in non-consistent states. More precisely, in APART$^+$, the back-end data server discards any output event e optimistically generated by a sink in round r (of the coordination phase), if the state globally imposed in round $r-1$ differs from the state reached by that sink at the end of round $r-1$. We call this class of optimistic output events *illegal* and, conversely, term as *legal* any output event that is either conservative or optimistic but not illegal.

Once the back-end data server gathers $minProposals$ legal output events, this selects one of them, and broadcasts it back to the sink replicas, along with the corresponding FSM state and vector clock. At this point, when a sink replica receives a decision message for round r of the coordination phase, it verifies if the vector clock and the (digest of the) FSM state selected by the coordination scheme coincide with those locally maintained when the output event was generated for round r. In the positive case, the sink replica can just keep on processing the incoming data streams, without any performance penalty. Contrarily, replicas that detect a misalignment with respect to the outcome of a coordination phase, re-install a globally consistent state. Then, they re-process any sensor message that had been already optimistically processed.

Note that state comparison and transfer is based on the digest representation just to reduce the communication delay, the storage requirements (to store state histories at the sink side) and to allow scalability of operations at the back-end tier.

3.3 Sink Behavior

Figure 1 shows the pseudo-code for the behavior of sink processes, single threaded for presentation simplicity. The sink maintains the following data structures: i) *msgBuffer* used to buffer incoming messages, which is assumed to provide FIFO semantic; ii) a vector clock *VC*, keeping track of communication histories with sensor processes; iii) two sequentially increasing counters, *stableRId*, and *optRId*, used, respectively, to identify the latest coordination rounds in which the sink externalized a conservative and an optimistic output event; iv) a boolean variable *optimisticMode* whose value reflects whether there are pending coordination phases, in whose case new output events, if any, are to be considered optimistic; v) an array of digests *SinkStateHist*, which is indexed via coordination round identifiers and whose entries store a compact fingerprint of the FSM state and vector clock at the time in which the sink generated an output event for a given coordination round.

If the sink process receives a message from a sensor process, independently of whether there are still pending coordination phases, the local vector clock is used to detect whether the sensor message has already been incorporated into the execution history currently seen by the sink ([1]). In the positive case the message is simply discarded. Otherwise the sink buffers the message, updates its vector clock to reflect the message reception and, by calling the `HandleInput` function, 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 first computes the digest of its FSM state and vector clock through the `hash` primitive. Then, depending on whether there are currently on going coordination phases or not, the sink generates either an optimistic output event, or a conservative one. In the latter case, it delivers the FSM output event to the back-end data server by means of a STABLE-OUTPUT message, piggy-backing the current stable output event identifier, namely *stableRId*, the state of the local FSM, retrieved via the primitive `getFSMState`, the local vector clock, and the current state digest. Also, the sink aligns the optimistic round identifier with the *stableRId* and flags the *optimisticMode* variable to signal the existence of active coordination phases. On the other hand, if the generated output event is an optimistic one, the sink sends out an OPTOUTPUT message, which is tagged not only with the

[1] This may happen if the sink has installed the state of a different replica, which already received and processed that message.

```
FIFOQueue msgBuffer;                                   // FIFO ordered message buffer
VectorClock VC;                                        // sensor messages history
int stableRId=0;                                       // stable round identifier
int optRId=0;                                          // optimistic round identifier
boolean optimisticMode=false;                          // current processing mode
array of Digest SinkStateHist={⊥,. . .,⊥};             // stores hashes of sink states externalized but not yet validated

upon receive(SENSORMSG, msgId, data) from sensor_i do
    if (msgId > VC[sensor_i])                          // filter out obsolete sensor messages
        VC[sensor_i] = msgId;                          // update the corresponding vector clock entry
        msgBuffer.push([msgId,sensor_i,data]);         // buffer the incoming message
        HandleInputMsg(msgId, data);

upon receive(DECISION, roundId, FSMState, vectorClock) from back-end server do
    if (roundId ≥ stableRId)                           // filter out obsolete coordination messages
        // unset the retransmission timers of all the output events generated up to the roundId-th coordination round
        ∀ Msg m where (isSetRetransmissionTimer(m) ∧ m.roundId≤ roundId) do unsetRetransmissionTimeout(m);
        stableRId = roundId;                           // set round counters
        clearBuffer(vectorClock);                      // prune message buffer
        if (stableRId=optRId) optimisticMode=false;    // enter non-optimistic mode
        if (SinkStateHist[roundId]≠ hash(FSMState,vectorClock))  // local state requires re-alignment
            setFSMState(FSMState);                     // align FSM internal state
            VC = vectorClock;                          // align local vector clock
            optId=stableRId;                           // align optimistic and stable round id
            optimisticMode=false;                      // enter non-optimistic mode
            while (msgBuffer ≠ ∅) do                   // re-process buffered messages
                [msgId,sensor_i,data]=msgBuffer.pop();
                HandleInputMsg(msgId,data);
        ∀i<stableRId do FSMStateHist[i]=⊥;             // remove obsolete FSM state hashes

void HandleInputMsg(int msgId, SensorData data)
    Msg m;
    OutputEvent outEv=ProcessMessage(data);            // update local FSM
    if (outEv ≠ null)                                  // the FSM producds an output event
        Digest dgst=hash( getFSMState(), VC );         // compute hash of the current sink's state
        if (¬optimisticMode)
            stableRId++;                               // increase stable round id
            optRId=stableRId;                          // accordingly align optimistic round id
            m=[STABLEOUTPUT, stableRId, outEv, getFSMState(), VC, dgst];
            optimisticMode=true;                       // enter optimistic mode
        else
            // get hash of FSM state at the time of the last output generation;
            Digest prevDgst=SinkStateHist[optRId];
            optRId++;                                  // increase optimistic round id
            m=[OPTOUTPUT, optRId, outMsg, prevDgst, getFSMState(), VC, dgst];
        send(m) to back-end server;
        setRetransmissionTimeout(m);                   // set rentransmission timeout for m
        FSMStateHist[optRid]=dgst;                     // store hash of the current sink's state

void clearBuffer(VectorClock stableMsgs)               // message buffer pruning
    ∀msg ∈ msgBuffer where msg.id ≤ VC[msg.source] do msgBuffer.remove(msg);

upon timeoutExpired(Message m) do
    send(m) to back-end server;
    setRetransmissionTimeout(m);                       // set the retransmission timeout for m
```

Figure 1. Sink Process Behavior.

local FSM state, vector clock and digest, but also with the digest associated with the last pending coordination phase and the current $optRId$ value.

Independently of whether the generated output event was optimistic or conservative, the sink stores the digest of its current state in the *SinkStateHist* array's entry associated with the just generated output event. Finally, in order to ensure the termination of the coordination phase despite crashes of the back-end data server, the sink sets a retransmission timer to periodically re-transmit the output message to the back-end.

If a DECISION message for the coordination round $rounId$ is received from the back-end server, the sink first makes sure that this is not an obsolete message (associated with some obsolete coordination round). Then it unsets the timeout used to retransmit any output event message generated in any coordination round up to (and including) $roundId$. Note that a sink that is relatively slow with respect to the other replicas may receive DECISION messages notifying the outcome of coordination rounds that it did not yet activate, but that were triggered by some other faster replica. To cope with such situations, the sink aligns its *stableRId* with the $roundId$ conveyed by the DECISION message (rather than just sequentially increasing it). Then the sink prunes out of its buffer any message whose processing was already incorporated in the processing history associated with the output event selected by the coordination phase. Next, by comparing the $optRId$ and the $stableRid$ values, the sink determines whether the DECISION message notifies the conclusion of the last coordination phase acti-

vated by that same sink. If this is true, it means that the reception of this DECISION message will enforce the consistency (assuming there had actually been any disalignment) of the current sink state. Hence, since the next output event to be generated by the sink is necessarily a legal one, the *optimisticMode* flag is unset, signalling that the conservative processing mode was entered. At this point, the sink detects if it needs to re-install the state of some other replica, whose output event was selected by the $roundId$-th round of the coordination phase. As already hinted, this can be efficiently achieved by comparing the digest of the sink's state stored in the $roundId$-th entry of the *SinkStateHist* array with the digest of the FSM state and vector clock carried by the DECISION message. If a disalignment is detected, the sink updates the local FSM state and its local vector clock according to the payload of the received DECISION message. In this case, de facto, the sink is discarding the effects of the optimistic processing of any sensor data. Hence it can safely enter the conservative processing mode, and start processing any sensor messages left in its buffer. Finally, the locally maintained digests of the sink state for all the coordination rounds up to (and including) the $roundId$-th one are discarded, so to ensure timely removal of unneeded state logs at the sink.

3.4 Back-end Data Server Behavior

```
set sinkSet = {sink_1^R,. . .,sink_m^R};
Set proposals = {};
int curRoundId = 1;
Digest curStateDig = ⊥;

upon receive(OPTOUTPUT,rId,outMsg,prevDgst,FSMState,VC,newDgst)
      from sink_i ∧ curRoundId=roundId
   if ( prevDgst=curStateDig )
      decide(rId, outMsg, FSMState, localVC, newDgst);

upon receive(STABLEOUTPUT,rId,outMsg,FSMState,VC,newDgst)
      from sink_i^R ∧ curRoundId=roundId
      decide(rId,outMsg,FSMState,VC,newDgst);

void decide(int rId,Msg outMsg,FSMState s,VectorClock VC,Digest d)
   proposals.add([outMsg, s, VC, d]);
   if | proposals | ≥ minProposals do
      [selOutMsg, selState, selVC, selDig]= select(proposals);
      log([curRoundId, selOutMsg, selState, selVC, selDig]);
      send(DECISION,curRoundId,selState,selVC) to all sink_i^R ∈ sinkSet;
      PublishTransaction(curRoundId,outMsg);
      proposals={};
      curRoundId++;
      curStateDig=selDig;

upon recoverFromCrash do
   if (([roundId,outMsg,FSMState,VC,digest] = readFromLog())≠ ⊥)
      send(DECISION,roundId,FSMState,VC) to all sink_i^R ∈sinkSet;
      PublishTransaction(roundId,outMsg);
      curRoundId=roundId+1;
      curStateDig=digest;
```

Figure 2. Back-end Data Server Behavior.

Figure 2 shows the back-end data sever pseudo code. 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; (ii) a set, namely $proposals$, used to gather the legal output events generated by the sinks for the current round; (iii) the digest of the sink FSM state and vector clock associated with the latest accepted output event, maintained by the $curStateDif$ variable.

In the case of receipt of a conservative output event via a STABLEOUTPUT message, since this is guaranteed to be representative of a linearizable processing history, the event is directly included in the $proposals$ set. On the other hand, if an OPTOUTPUT message associated with the current round is received, it is first of all checked whether the corresponding output event is legal. To this end, the digest of the sink FSM state and vector clock at the time of generation of the former output event (i.e. at round *curRoundId-1*), which is piggybacked on the OPTOUTPUT message, is compared with the one locally maintained in $curState$. If the two digests do not match, the event is simply discarded, otherwise (i.e. if the optimistic output event is found to be legal) the event is inserted in the $proposals$ set.

As soon as the cardinality of the $proposals$ set reaches the $minProposals$ value, the data server invokes the `select` primitive to choose the output event selected as representative for publication (as well as the corresponding sink replica FSM state and vector clock), logs the choice on stable storage and sends back the decision (i.e. the result of the selection) to the sink replicas. Then it invokes `PublishTransaction` 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, increments the round counter and stores the digest of the selected sink replica state in *curState*.

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 re-executed multiple times (e.g. in the case of multiple subsequent crashes of the data server). Finally, the current round identifier and the digest of the latest stable sink's state are updated on the basis of the info retrieved from the log.

4 Related Work

APART$^+$ inherits the same advantages of APART [19] with respect to classical replication schemes (namely, those based on a-priori agreement on the processing order). Since these advantages have already been highlighted in the Introduction section, rather than focusing on a comparison with the state of the art of replication solutions (for which we remind the interested reader to [19]), in this section we discuss the relations between the APART$^+$ optimistic synchronization scheme and classical optimistic approaches in literature.

APART$^+$ allow multiple non-deterministic replicas to optimistically carry on processing activities based on potentially inconsistent initial state. Replicas speculate in

parallel on tentative execution paths, among which one is eventually selected as the committed path. In this aspect, APART$^+$ supports an execution model where logical time advances according to a committed horizon and an uncommitted one [3, 6]. This is typical of speculative high performance computing approaches, especially in the context of virtual reality and simulation systems. Compared to these approaches, one main distinguishing feature of APART$^+$ is the presence of reconciliation, typically not employed in synchronization schemes for high performance computing due to the fact that speculation operates on a per component basis (instead of across multiple replicas). Similar considerations can be made when comparing the APART$^+$ optimistic approach with speculative execution techniques used in pipelined computing architectures [13], or with optimistic concurrency control [1] in transactional systems. In the former scenario, optimistic branch predictions that are contradicted by later pipeline stages result in a re-execution of a different branch. In the case of optimistic concurrency control, analogously, the detection of a conflict due to incorrect speculative data accesses forces transactions to rollback, and (possibly) re-execute.

The aforementioned optimistic approaches are characterized by an inherent tradeoff. If the "optimistic" assumptions turn out to be valid, optimism pays off, providing a significant performance boost. However, if the optimistic assumptions do not hold, these protocols execute more slowly than a pessimistic one due to the costs imposed to rollback the system to a correct state. In other words, the gain in performance outweighs the overhead of repairing actions that execute incorrectly only if the optimistic assumptions hold frequently enough. Contrarily, in nice runs where no failures occur, the performance of the APART$^+$ optimistic synchronization scheme is in practice unaffected by the probability that the optimistic assumption actually holds. This depends on that, in absence of failures, at least one replica is never aborted by reconciliation.

5 Performance Evaluation

The performance evaluation study presented in this section is based on a process-oriented simulation model developed in JAVA2 on top of the JavaSim 0.3 library. Our analysis if focused on a failure free scenario, as this is typically the most frequent in practice. We consider a system composed of 10 sensors, 3 sinks, and one back-end data server. The *minProposals* parameter is set to 1, so to to minimize the end-to-end data processing latency.

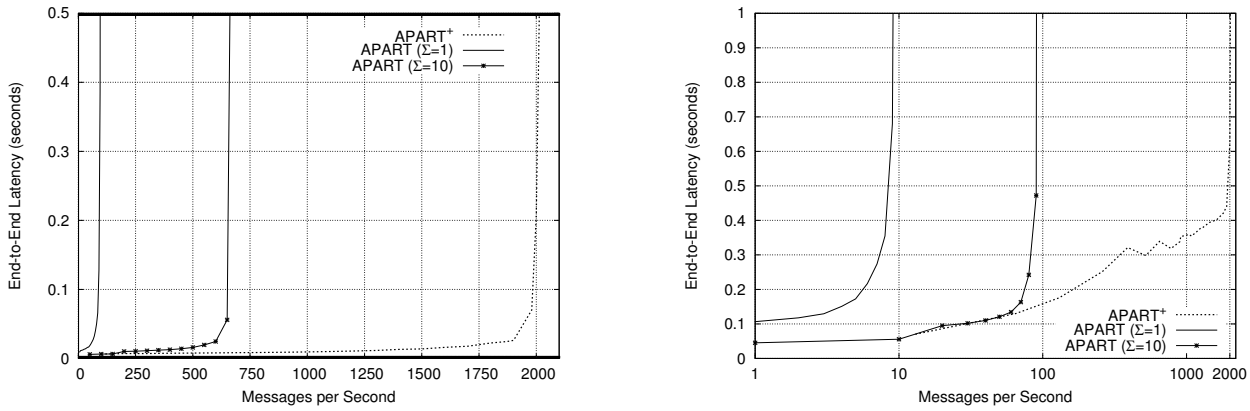
Each sensor acts as a poissonian source, generating inputs for the sinks with an average interarrival rate which is treated as an independent parameter. The sinks and the back-end data server process incoming messages with exponential service rates, whose mean values are, respectively, 2000 msgs/sec and 20000 msgs/sec. The choice of these parameters is intentionally aimed at preventing the back-end data server from becoming the system's bottleneck. Such a choice allows us to identify the upper bound on the throughput achievable by the replica coordination schemes employed by APART and APART$^+$, which is the actual focus of this study, i.e. we exclude any performance limiting effect caused by the saturation of the back-end server. Further, in large scale data acquisition systems, the back-end is typically deployed over carefully dimensioned clustered systems in order not to represent a performance bottleneck, as well as to ensure its high availability.

To evaluate how the two considered protocols fare in the presence of diverse application level logics at the sink's side, we rely on two independent parameters: i) Σ, namely the sink's "silentness-degree", (see Section 2.2), and ii) p_{abort}, namely the probability for the back-end data server to send Decision messages informing a sink (whose output event is not selected during a synchronization phase) to install the state of some other replica.

In our study, we consider two typical deployment scenarios, characterized by different inter-process communication latencies. In the first one (namely the *LAN* scenario), components are distributed over the same local area network and message delivery latencies are assumed to be exponentially distributed with a 5 milliseconds' mean value. In the second one (namely the *WAN* scenario) components are deployed over a wide area network and the mean communication latency (one-way) is set to 50 milliseconds.

Figure 3(a) and Figure 3(b) report the average end-to-end latency (evaluated as the time interval since the production of a sensor message and the publication of the corresponding output event at the back-end data server) for the two scenarios. The plots report experimental results obtained by setting p_{abort} to 0.5. However, it was found out that the performance of both protocols are only negligibly affected by the probability of replicas to undergo a state re-installation upon completion of a reconciliation phase. This depends on that, in both protocols, at least one replica is never aborted in each synchronization round, and hence does not incur any performance penalization. The simulation results also highlight that, in APART$^+$, a sink that succeeds in imposing its state in early coordination rounds accumulates a remarkable advantage over other replicas while optimistically processing input data (this phenomenon was particularly evident at high load).

For the APART protocol we report performances when considering both the cases of: i) a non-silent FSM, producing an output event per each incoming sensor message ($\Sigma = 1$), and ii) a FSM producing an output event each 10 processed sensor messages ($\Sigma = 10$). The Σ value, in fact, has a strong impact on the frequency of activation of the replica synchronization phase, which represents the APART's bottleneck for the considered parameters' settings. This is confirmed by the plots in Figure 3, which clearly show that an increase in the FSMs' silentness-degree causes a proportional increase of the maximum sustainable throughput in APART. On the other hand, the performances of the APART$^+$ protocol are almost independent of the Σ parameter (which is the reason why we here report only the



(a) LAN Scenario (b) WAN Scenario

Figure 3. Average End-to-End Message Processing Latency.

results obtained with $\Sigma = 1$). In APART$^+$, in fact, since the synchronization phase is not blocking, the sinks' processing activities are never halted, even in the presence of pending synchronization phases. Hence, the only cost to be affected by variations of the Σ parameter is related to the corresponding alteration in the frequency of generation of output messages at the sink. However, in our simulation we do not explicitly model the costs associated with sending output events at the sink (in terms of both local processing activities and network bandwidth consumption), which are assumed to be negligible when compared to the sensor messages' processing latency ([2]).

Overall, by the plots in Figures 3(a) and 3(b), the blocking synchronization strongly hinders APART's performance, especially in the WAN scenario, whose higher communication latencies imply a longer duration of the synchronization phase, limiting its maximum throughput to 10 msgs/sec for $\Sigma = 1$. On the other hand, the optimistic pipelining of output events in APART$^+$ reveals extremely effective, permitting to achieve, in both the LAN and WAN scenarios (and independently of the sinks' silentness degree) a maximum throughput that is just slightly lower than the theoretical bound in absence of replication overheads for the considered parameters' settings.

6 Conclusions

In this paper we have shown how the adoption of optimistic processing techniques can significantly boost the performance of a recently proposed active replication scheme, called APART. Through a simulation based study, we have shown that, in nice runs, the performance of the APART$^+$ optimistic synchronization scheme is not affected by the probability that the optimistic assumption holds. This depends on that, in each reconciliation phase, there is at least one replica that is never aborted, which can steadily advance in the processing of incoming data with no or very limited performance penalization.

[2]This assumption is introduced primarily to simplify the simulation model, but the approximation error is expected to be small if the size of the output events' messages is relatively small, which is often true in practice.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing, 1987.

[2] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comp. Surveys*, 36(4):372–421, 2004.

[3] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

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

[5] 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.

[6] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

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

[8] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS)*, page 156. IEEE Computer Society, 1998.

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

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

[11] National Institute of Standards and Technology. Secure hash standard, 2002.

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

[13] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[14] P. R. H. Place and K. C. Kang. Safety-critical software: status report and annoted bibliography. Tech.report, CMU/SEI-92-TR-5 (ESC-TR-93-182), 1993.

[15] A. Rahmati, L. Zhong, M. Hiltunen, and R. Jana. Reliability techniques for rfid-based object tracking applications. In *Proc.of the 37th Conference on Dependable Systems and Networks (DSN)*, pages 113–118, 2007.

[16] A. Rahmati, L. Zhong, M. Hiltunen, and R. Jana. Reliability techniques for rfid-based object tracking applications. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 113–118, Edinburgh, UK, 2007. IEEE Computer Society.

[17] F. Quaglia and P. Romano. Ensuring e-Transaction with asynchronous and uncoordinated application server replicas. *IEEE Transactions on Parallel and Distributed Systems*, 18(3), 2007.

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

[19] P. Romano, D. Rughetti, F. Quaglia, and B. Ciciani. Apart: Low cost active replication for multi-tier data acquisition systems. In *Proc. of the IEEE Symp. on Network Computing and Applications (NCA)*, pages 1–8, 2008.

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