Multithreading-Enabled Active Replication for Event Stream Processing Operators

Andrey Brito, Christof Fetzer
Systems Engineering Group
Technische Universität Dresden
Dresden, Germany
Email: {andrey, christof}@se.inf.tu-dresden.de

Pascal Felber
Institut d'informatique
University of Neuchâtel
Neuchâtel, Switzerland
Email: pascal.felber@unine.ch

***Abstract*—Event Stream Processing (ESP) systems are very popular in monitoring applications. Algorithmic trading, network monitoring and sensor networks are good examples of applications that rely upon ESP systems. As these systems become larger and more widely deployed, they have to answer increasingly stronger requirements that are often difficult to satisfy. Fault-tolerance is a good example of such a non-trivial requirement. Making ESP operators fault-tolerant can add considerable performance overhead to the application. In this paper, we focus on active replication as an approach to provide fault-tolerance to ESP operators. More precisely, we address the performance costs of active replication for operators in distributed ESP applications. We use a speculation mechanism based on Software Transactional Memory (STM) to achieve the following goals: (i) enable replicas to make progress using optimistic delivery; (ii) enable early forwarding of speculative computation results; (iii) enable active replication of multi-threaded operators using transactional executions. Experimental evaluation shows that, using this combination of mechanisms, one can implement highly efficient fault-tolerant ESP operators.**

I. Introduction

Event Stream Processing (ESP) systems target applications that produce data continuously and requires (soft) real-time processing. An ESP application consists in a directed acyclic graph of operators that are traversed by the data. Inside each operator, the pieces of data, called events, can be transformed, aggregated, combined, enriched, or filtered out. A common application scenario is when too much data is generated and storing the complete data set is infeasible, like in large sensor networks, subatomic physic experiments, or even network monitoring. In such cases, a huge amount of low-level events should be summarized into fewer but more meaningful events that will be later processed using traditional techniques (e.g., databases). Similarly, there are cases when the data should be analyzed on-the-fly and its value is inversely proportional to the processing time, e.g., algorithmic trading, and traditional store-and-process approaches impose unacceptable latencies. Clearly, besides real-time performance, failures are not desirable in either case: in the former, important information may be lost; in the latter, missed opportunities directly translate into financial losses.

In this work, we investigate the use of active replication in ESP applications. Active replication [1] and the state machine approach [2] aim at masking failures by the replicating operators and letting the replicas work independently. Thus, a failure in one of the replicas can be transparently masked by the other replicas. Unfortunately, operators are frequently dependent on a persistent local state. To ensure consistency among the replicas, events must be processed in the same order at different replicas and computations must be deterministic. The immediate consequence of this requirement is the usage of an ordered message delivering mechanism (e.g., atomic broadcasts [3]), which adds latency by requiring an agreement on the order of the messages. Even after events have been received in the same order at different replicas, execution must still preserve this ordering because the processing order can have an effect on the local state. Therefore, multi-threading is typically not possible because it introduces non-deterministic state changes. These two restrictions impose a considerable performance cost for the usage of active replication. Addressing these two restrictions is the focus of this paper.

Fault tolerance is a well studied topic and different approaches exist, each providing different trade-offs. Passive replication [4], for example, considers that a primary replica logs state changes in stable storage. When the primary replica fails, a backup replica takes over; it restores the state from stable storage and proceeds with the computation. Although this approach does not require an atomic broadcast, the logging of the state changes may also add considerable latency. In addition, the recovery process requires the backup replica to rebuild the state since the last checkpoint, adding extra delays that were not needed by active replication. A clear advantage is that inputs are processed only in one replica, thus, less resources are used. Another advantage is that passive replication allows non-deterministic decisions taken during the processing of the input (e.g., the scheduling of concurrent threads) to be included in the log.

Semi-passive [5] and semi-active [6] replication are two

other approaches. They basically try to combine benefits from both passive and active replication. In semi-passive replication, a primary node performs the processing and sends the state changes to a backup replica that immediately applies them, thus allowing for fast failover in case the primary fails. Having the primary process the input and send only the state changes requires few processing resources from the replica. In semi-active replication, a leader processes the inputs first and sends all relevant decision to the replica, for example, the ordering of the inputs it processes and the scheduling decisions (in case of multi-threading). In this latter approach, the replica's state is close to that of the primary and thus recovery is fast. Nevertheless, in both cases, each message sent by the primary (either the state changes in semi-passive replication or the guiding decisions in semi-active replication) requires an acknowledgement by the replica before the primary is allowed to produce an output. This round-trip communication delay under normal operation is inevitable because replicas are not independent. These approaches then do not replace active replication.

The challenges discussed above are also present in ESP systems. Some systems (e.g., Streamflex [7]) consider only stateless operators and thus active replication is trivial. FLUX [8] and Borealis [9] are examples of active replication for stateful components, but they only apply the traditional approaches discussed above.

In this work, we present the novel approach used in StreamMine [10], [11] to provide low overhead active replication in ESP applications. The key component of our solution is the usage of a speculation mechanism based on Software Transactional Memory (STM) [12]. STM enables efficient and fast rollback when the speculation is not successful. Using STM, we can achieve the following contributions: (i) enable replicas to make progress based on optimistic deliveries while waiting for the final ordering of messages; (ii) enable nodes to forward speculative information and downstream nodes to pre-process these speculative inputs; (iii) enable multi-threaded processing in operators using transactional executions.

The rest paper is organized as follows. In Section II, we introduce important background information, as well as our system model and an example that we will use throughout the paper. Then, in Section III, we present how we use the speculation mechanism to implement active replication. An evaluation of our approach is presented in Section IV and in Section V we discuss related work. Finally, we conclude in Section VI.

II. Background and System Model

A. ESP application example

In this section we detail a simple application example that will be used to explain the concepts throughout the paper. In our example, depicted in Figure 1, we consider two sources of events (labeled `Publisher` in the figure). These sources generate events containing, for example, stock quotes. The events are then analyzed by an operator (`Processor`) that issues events encapsulating actions based on the current as well as past input events. Note that the actions issued may be different if two events are processed in a different order. Next, information from outside sources that are needed in order to fulfill the action are added by the `Enrich` operator. This operation may be expensive if it accesses external resources (e.g., look up an id in a database), but can be parallelized by simple replication as the operation depends only on the current input. Finally, the `Split` operator distributes (e.g., for load-balancing) the events between two `Consumer` components, which perform the actual execution of the action described.

In order to make this application fault tolerant, the most fundamental step is to ensure that failures of the operators can be masked in a precise and timely manner. The `Enrich` and `Split` operators are stateless and, therefore, can be trivially replicated for fault-tolerance. In addition, their replicas would process inputs deterministically and generate exact copies of the output events, which simplifies elimination of duplicates by the succeeding operators. For this reason, we omit their replications in Figure 1.

Consider now the `Processor` operator. Assume initially that this operator is sequential, i.e., only one thread processes input events. Even in this case, results would not be correct if we simply replicate it. The reason is that input events generated by the publishers could arrive at the two replicas in different orders. As a consequence, decisions triggered by these events could be different (e.g., if the first event that arrives consumes some resource). These differences would propagate to downstream nodes, which would not be able to cope with the two different views. Note that even if one of the replicas is prioritized and acts as primary, upon failure the other replica would take over with a state that is potentially completely inconsistent from the viewpoint of the downstream nodes. An ordered reliable communication mechanism is thus needed to guarantee that both replicas receive the same events in the same order.

Assume now that the `Processor` operator is parallelized. Several threads are available and each of them picks an event from the input, executes some computation that modifies the local state, and generates an output. If these threads access shared memory locations, these accesses must be protected by, for example, locks. Thus, the order of acquisition of the locks may directly affect the computation and the operator state. Nevertheless, to conciliate these scheduling decisions on both replicas they need to communicate. This communication adds latency as a replica cannot output a result before receiving an acknowledgement from the other replica. As an example, if one replica outputs an event and fails before the other learns the lock acquisition sequence for that output, it is likely that the surviving replica will have a different execution and reach a different result. Once

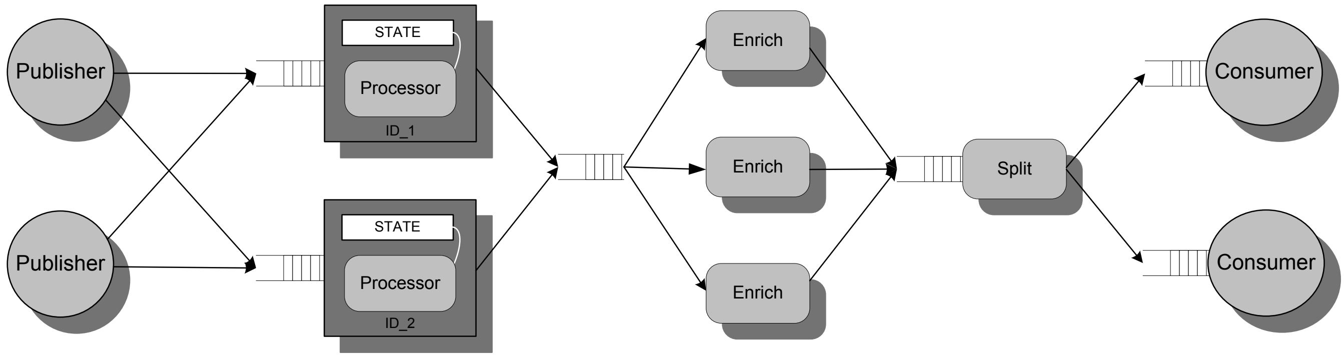


Figure 1. A simple prototypical ESP application graph with replicated components.

again, the downstream node would face an inconsistency. If lock-free algorithms are used to enable higher parallelism, ensuring repeatability becomes event more complex.

Another small, but highly practical, hurdle in parallelizing operators is that the implementation of parallel code is far from trivial. A parallel component requires concurrency control to guarantee that memory positions that are accessed by more than one thread are protected from race conditions. In addition, the state must be kept consistent (e.g., a thread must be protected from seeing intermediary computation state from other threads). Achieving high parallelism requires fine grained control. With locks, for example, it requires locking only when strictly necessary and releasing locks as soon as possible. For this reason, bugs happen much more often in concurrent code. In our example, even benign concurrency bugs (e.g., bugs that cause an incorrect, but acceptable result) can lead to an inconsistency between the two replicas.

B. A software transactional memory for speculative event processing

The key component of our approach is an STM-based speculation mechanism. Here we provide a brief overview of the provided features. For detailed discussion about the implementation and the overheads involved, we refer the reader to [13] and [14].

Our work is based on TinySTM [15], an open-source STM implementation in C. STM was introduced as synchronization mechanism that is easier to use and potentially more scalable than locks. Consider a fragment of code that accesses shared memory. With locks, this code would be surrounded by lock acquisition and release and it would be guaranteed that only one thread executes this *critical section* at a time. With STM, the fragment of code would be surrounded by the start and end delimiters of the transaction. Threads may then execute the critical section concurrently and the STM monitors their memory access directed at shared memory locations. If these threads indeed access the same positions, they conflict and one of them may need to roll back and re-execute (i.e., *abort*). Otherwise, if their accesses do not collide (e.g., threads accessed independent fields of an object) both can complete the transaction (i.e., *commit*). Thus, instead of having the programmer explicitly lock individual memory locations in a conservative manner, STM provides automated fine-grained runtime control.

In order to provide the functionalities explained above, STMs are typically optimistic. In our case, TinySTM isolates the execution of a transaction by intercepting shared-memory accesses (i.e., reads, writes, memory allocations and releases) and redirecting writes to a reserved portion of memory, named *write set*. At the end of the transaction, the STM validates the execution, i.e., all reads and writes must be consistent so that the complete transaction appears to execute atomically. If validation is successful, buffered updates from the write set are applied to their original destinations and the transaction completes. We additionally impose an ordering on processing by enforcing transactions with lower timestamps to be committed before those with higher timestamps. This optimistic execution model is the base for our speculation mechanism.

In our case, the complete processing code for an input event is encapsulated within a transaction that uses the input ordering of the events as timestamp. In addition, a timestamp has a flag denoting if it is speculative or not. A speculative timestamp is assigned by an optimistic delivery and a non-speculative timestamp by the final delivery of an atomic broadcast message. When a transaction finishes processing, it validates, verifies that all transactions with lower timestamps have committed, and checks if it is already non-speculative. If the transaction validates correctly, but there are still non-committed transactions with lower timestamps or it is marked as speculative, the transaction enters a pre-committed state. It can only commit after all its dependencies are resolved (i.e., transactions with lower timestamps commit and the transaction becomes non-speculative). Otherwise, if for example the timestamp was

incorrect because of a wrong speculation, the transaction is aborted (discarding all buffered updates in the write set) and restarted with a new timestamp.

Another two important extensions that we added to TinySTM are modifications to the dependency tracking and optimism control mechanisms. Two ongoing transactions conflict when one writes to a position that the other has read or written. In our case, the transaction with the higher timestamp *depends* on the transaction with the lower timestamp. We can then allow ongoing transactions with higher timestamps to read from the write sets of the transactions it depends on if the latter are in pre-committed state. Subsequently, if the speculation of the lower timestamped transaction turns out to be incorrect and the transaction aborts, all other transactions that read from its write set must also be aborted. In order to bound the amount of resources that can be wasted by exaggerated speculation, we use a optimism control component named *conflict predictor*. The conflict predictor, detailed in [14], essentially limits speculation based on the average amount of rollbacks experienced at runtime.

Finally, it is important to notice that the instrumentation required in order to allow TinySTM to intercept shared memory accesses can be done automatically at compile time with TANGER [16]. In addition, destructive external accesses (e.g., writes to external devices) cannot be rollback automatically. Therefore, when such accesses are used inside an event processing function, the designer must choose between either explicitly providing undo functions that enables any wrong speculative access to be rolled back, or place the external access in a special *on-commit* block. This block is executed non-speculatively when all pending speculations are confirmed and the transaction is about to commit.

C. System model

The system model used in this work considers a system composed of ESP operators running on distributed machines and interconnected by reliable FIFO connections (e.g., TCP). After an operator fails, a new operator must be created so that the system remains fault-tolerant. To that end, we use a simulated perfect failure detector as detailed in [17]. This failure detector uses watchdogs (either software or hardware) and leases to force crashed nodes to be restarted. In addition, it transforms timing failures (i.e., nodes that are unresponsive because they are too slow) into crash failures.

In our system, during the application deployment, an operator marked as fault tolerant is replicated in two nodes (the replication level can be modified). The different replicas have the same specification, but are assigned different ids. When one of them fails and is restarted, it recovers by listening to messages while transferring the most recent (non-speculative) state from the surviving replicas.

The atomic broadcast protocol requires consensus. A classical consensus protocol (like the ones presented in [3]) is to use the surviving node with the lowest id as first coordinator and have it send a proposed value (i.e., the ordering of the messages) to the other nodes. For the case of two replicas or during failure free runs, this protocol can be efficient implemented with the perfect failure detector discussed above. Note that the atomic broadcast protocol is completely disconnected from the operators logic, other atomic broadcast protocols could also be used. Another important feature of the communication protocol to be used is optimistic delivery. When a message is received by a node, it is assigned a tentative order and is delivered optimistically to the operator. If the operator has speculation enabled it can proceed speculatively, otherwise it may ignore the message. When an agreement on the final order is reached (either by receiving the ordering or by suspecting the other replica in the case of two nodes), the message is delivered with a final ordering value that may or may not be the same as the optimistic order. We will show later that these two orderings are often the same and this can be exploited by the speculation mechanism.

When nodes receive an input event they may produce output events. If the input event has a *speculative timestamp*, any output event produced during its processing will be marked as *speculative event*. In contrast, if the input event is final (and there is no earlier speculation pending) the output events will be also final. When a transaction commits, output events that it has previously generated as speculative will be re-sent as final. Because speculative events are optimistically generated, there is no guarantee that they will be consistent between replicas or that future version of these events (e.g., the final, non-speculative, version) will carry the same content. On the contrary, final events outputted by an operator are produced as a result of both a final ordering (as opposed to an optimistic ordering) *and* final input events and are, therefore, guaranteed to be the same among replicas.

From the viewpoint of an operator placed downstream of a replicated operator, events have unique ids, which typically consist of tuples with the id of the operator that created the event, the replica id, and the number of final events that the replica has produced so far. Because final events produced by two different replicas will both have the same ids (except for the replica id subfield) and the same contents, the downstream component can use this id to detect and discard duplicates. Finally, we would like to mention one additional optimization we do regarding speculative events. For practical reasons, nodes only process speculative events originating from the lowest-id replica of an upstream replicated operator. An analysis of why this is helpful will be done in the next sections.

III. Speculation in active replication

In the previous section we gave an overview of how the high-level operation of the system, in this section we discuss in the details the usage of speculation in the context of active

replication. We first consider the cases of single and multi-threaded operators in failure free runs, and then we discuss the effects of failures.

A. Single-threaded operators

We initially consider the case of a single-threaded operator replicated on two nodes. A first possible improvement is to overlap the execution of the atomic broadcast with some useful computation. Recall that, as detailed in the previous section, the final ordering of the messages will be the ordering of one of the replicas, say replica R_1. The other replica, R_2, receives the ordering from R_1. Note also that it is desirable that both nodes have approximately the same speed, otherwise, the faster node would need to wait for the slower node to catch up.

There are three sources of potential gains here. First, the ordering of replica R_1 cannot be used as final as soon as the message arrives. This is a consequence of the fact that, if R_1 fails before communicating this ordering to replica R_2, but after generating an output event, the downstream operator could see an inconsistency. Nevertheless, if replica R_1 does not fail, its ordering will be preserved and optimistic delivery will be guaranteed to succeed. Thus R_1 can process its inputs speculatively, producing speculative output events, and after receiving the acknowledge from R_2, R_1 can commit its computations and send its events as final.

Second, the node running replica R_2 receives input events and the communication protocol delivers them optimistically. Immediately, R_2 can start processing this events speculatively. There are two scenarios where these speculations have high chance of success: (i) if the system is under low load, the variation in the arrival time of the events on different nodes will be lower than the inter-arrival times of the events, thus events will be naturally ordered; (ii) if the system is under very high load, for example, during a burst of events where the inter-arrival times is lower than the time needed to process them, the events will accumulate on the input queues, allowing them to be picked deterministically[1]. We will show in the next section experiments that confirm this intuition. Nevertheless, R_2 will sometimes speculate incorrectly, but R_2 learns the final order first (at the end of the first consensus round) and, thus, has more time to rollback and re-execute computations.

The third and last potential gain relates to speculative events. Consider replica R_1. This replica starts processing events speculatively as soon as they arrive and, unless it fails, this speculation will succeed. As a result of the processing, R_1 may generate speculative events. Once again, if this replica does not fail, these speculative events will have the exact same content of the final events. Thus, if these events are forwarded downstream, the next component can already process them (also speculatively). Later, if no failures occurs, the final events will be identical to previously sent speculative events and will not need to be reprocessed. An additional gain is then achieved by overlapping computation on downstream nodes with the atomic broadcast. Recall from the previous section that the communication middleware will discard speculative events from replicas other than the one with the lowest id. This is an interesting optimization because it both results in lower network load and reduced overhead on the downstream nodes by allowing them to only consider speculative events that are most likely to be identical to final ones.

In summary, it is possible to overlap the time required for consensus with useful processing. This gain can be particularly important in application scenarios where replicas are placed in geographically or logically distant nodes. In such scenarios, the distance will lead to communication latencies potentially much higher than the processing times. Notwithstanding, we show in the next section that these gains are considerable even for applications in local clusters.

B. Multi-threaded operators

One serious limitation of active replication is that it cannot easily support multi-threaded operators. Typically, to support multi-threading, one would use a semi-active or semi-passive replication approach and the scheduling decisions will be transmitted from one of the replicas (usually named leader) to the others (the followers). This approach is expensive and complex. First, each synchronization operation (e.g., lock acquisition and release, or atomic operations used by lock-free algorithms) must be distributed among replicas. Second, the communication of the scheduling decisions also requires consensus and, thus, introduces additional latency.

In the case of our STM-based speculation, the many possible interleavings of concurrent threads need to be neither communicated nor guessed as the STM causes executions to appear as atomic. As a consequence, all the potential scheduling interferences will be masked away by the STM and can be uniquely determined by the ordering of the messages in the processing sequence, exactly as in the single-threaded case.

It should be noted that STM is typically not as efficient as optimized parallel code (lock-free or based on fine-grained locks) due to the the overheads of instrumented memory accesses. Even then, STMs are useful because they greatly simplify the process of developing parallel code. In our case, once we consider the costs of registering the decisions and propagating them, the overhead of STM is quickly outweighed.

C. Dealing with failures

As discussed in earlier sections, the greatest advantage of active replication is the ability to mask failures transparently.

[1]Deterministic ordering in this case can be implemented by having one single priority queue for all events from all connections arriving at an operator, and ordering the events by their ids.

Although this is true at a high level, because active replication depends on an underlying consensus for the atomic broadcast, failures need to be considered at the lower levels.

In our system, we use a simulated perfect failure detector as described in [17]. Nodes are equipped with local clocks with drift rates bounded by ρ_{max}, and with hardware or software watchdogs (e.g., the ones provided as kernel modules in all common Linux distributions). The watchdogs will cause a node to be restarted unless it manages to acquire a lease from one of its peers every T time units. For example, in case of three nodes, if a node N_1 did not grant in the last $T * (1 + \rho_{max})$ a lease to node N_2 or to a node N_3 that granted a lease to N_2, then it knows for sure that N_2 has failed and is being restarted.

From the viewpoint of the replica with the lowest id, say R_1, which is the first consensus coordinator, a failure of the other replicas has little effect. R_1 will speculatively process messages in an order that will become final, then as soon as it detects the failure of the other replica (in the case of two replicas) it can finish the second round of the consensus and commit its computations.

From the viewpoint of the replica with the second lowest id, say R_2, it also processes speculatively messages as they arrive. Next, as soon as it detects a failure of R_1, it can stop waiting for the proposed ordering and use its own. If only two replicas are used, R_2 can immediately commit all its speculations. For the case of multiple replicas, R_2 would become the new coordinator and would start a new consensus with its ordering as proposed value. Nevertheless, because R_2 has less confidence on the final order of the messages, it will tend to do less speculation (this mechanism is transparently controlled by the conflict predictor as mentioned in Section II). Therefore, a failure of the replica with the lowest id will tend to be more costly.

Finally, once a crashed (or slow) replica is forced to restart by its watchdog, it executes the recovery procedure: (i) it gets a new unique id and its operational status starts being monitored by the failure detector; (ii) it starts storing the messages it receives and also the final orderings from the consensus executions; (iii) it retrieves the committed state (i.e., the state that considers only final events) from the other replicas. These steps can be executed in parallel. Also note that the committed state for a timestamp ts will be the same in all replicas, which can be used to speed up state transfer. In addition, if necessary, a recovering replica can slow down the other replicas by delaying its acknowledgements in the consensus.

IV. EVALUATION

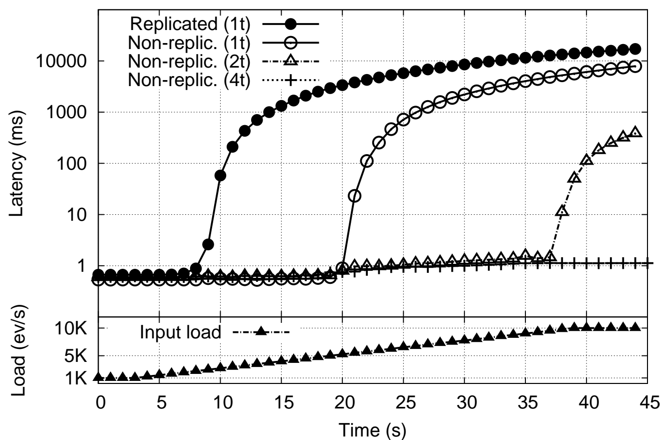
In this section, we evaluate the system as follows.

1) We compare the performance of the non-speculative, non-replicated versions (single and multi-threaded) of a typical stream operator with a non-speculative, replicated version. The goal is to determine the practical cost of replication.
2) We compare the speculative versions of the replicated operator (consuming different amounts of resources) with the non-speculative, replicated and the non-speculative, non-replicated version. This analysis evaluates if a speculative operator can indeed be a substitute for a non-replicated version.
3) We validate the assumption that events will often arrive in the same order in different nodes.
4) We finally evaluate the gains from forwarding speculative events early, both with and without failures.

Our experiments were executed using two Sun Niagara machines (running the operators) and two Intel Xeon (as publishers and consumers of the events), all of them running Linux.

We start our evaluation by showing the costs of introducing replication in a running system. For this purpose, we consider a stream processing operator that uses *sketches* to keep a summarized representation of the past events (for example, the count sketch operator from [18]). In such an operator, when a new event is received only parts of the state need to be read or updated. We built a sequential and a parallel version of this operator. The parallel version was built using locks with the finest granularity possible for maximum parallelism (it locks only the exact position of the state that is being accessed). We then deployed an ESP application similar to the one depicted in Figure 1. The response of the system to an increasing input load in terms of latency and throughput is shown in Figure 2 for the replicated and non-replicated versions. The sequential and parallel implementations were used for single-threaded and multi-threaded experiments, respectively. The input load is shown in the lower part of the left graph. It is kept constant in the first and in the last 5 seconds of execution (at $1,000$ and $10,000$ events/s, respectively). The difference between the non-replicated and replicated versions represents the practical cost of classical replication, i.e., without using speculation nor multi-threading.

Next, in Figure 3 we show the effects of speculation and multi-threading in the replicated component for the previous scenario (same input load). Note that the code running in the speculative version does not require manual parallelization. Therefore, by simply enabling speculation and allowing the speculative operator to execute three threads we can achieve the same throughput as the non-replicated operator and a throughput twice as high as the non-speculative replicated operator. Regarding latency, our current prototype does not allow speculation on the ordering when multiple threads are used. For this reason, the latency for low loads is higher with speculation. As we will show next, speculating on the order has a high potential and can also be useful with single-threaded speculative operators by enabling the early forwarding of events.



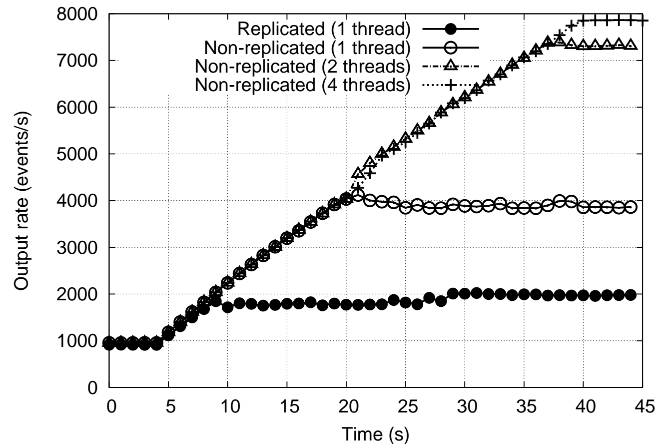
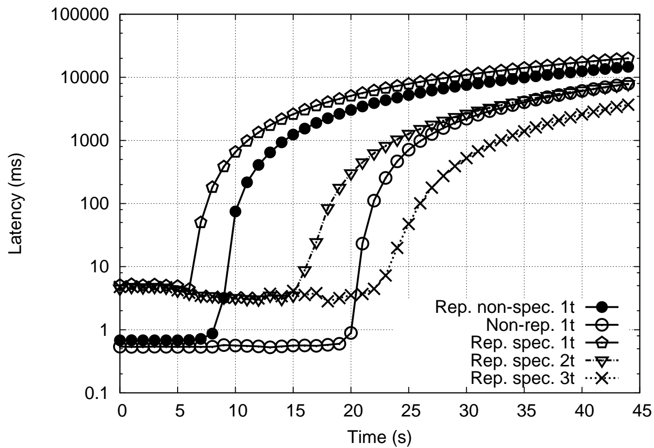


Figure 2. System response to different input loads with a replicated and several non-replicated versions of our operator without speculation.



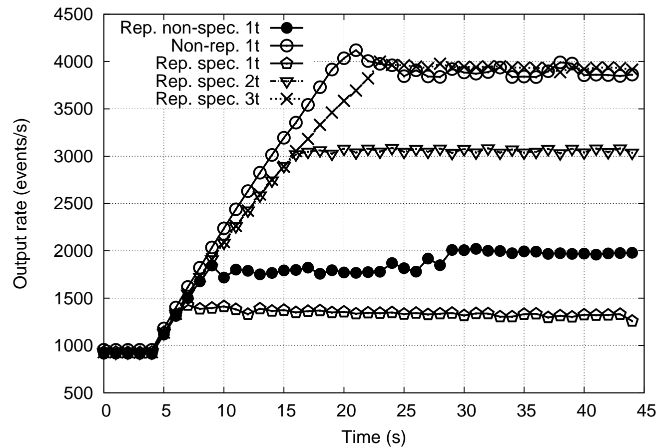


Figure 3. System response to different input loads with replicated versions of our operator with and without speculation.

In the sequence, we evaluate our assumption that messages sent by two different sources will often arrive at the two processors in the same order. In this experiment, each source emits events at increasing rates from 500 to $15,000$ events/second. The sources and the processors are placed on different machines in the same LAN. We then measure the rate of successful commits, rollbacks (i.e., misspeculations) and aborts (i.e., tasks that have been processed, but need to be discarded and reprocessed). These measurements are shown, respectively, in the upper, middle and lower part of Figure 4. In this experiment we use an empty operator (no computational costs) and consider the worst case for speculation, i.e., when misspeculation occurs and an event must be aborted and reprocessed, all the events that were speculatively processed after it will also need to be aborted and reprocessed. As discussed in Section II, for workloads with high parallelism, events that did not used parts of the state that were modified by the aborting event would not need to be aborted. As shown in the figure, the number of aborts (both absolute and relative) is higher in the area around time 30. This is consequence of two different effects. First, with lower input rates the variation in communication delays is lower then the interarrival times. Second, with high loads the messages accumulate in the input queues of the operators and allow deterministic ordering.

Our next step was to evaluate the potential benefit of the early processing and early forwarding of events. As shown in Figure 5 speculative events are received by the downstream operator around 100 μs earlier than the final events. Thus, speculation not only helps the current operators to start its processing earlier, before knowing the final ordering, but even enables downstream operators to benefit. If the next operator also has a lengthy computation (like the `Enrich` operator in Figure 1), being able to initiate the computation earlier reduces the end-to-end latency. Also note that the value of 100 μs is function of the communication latency,

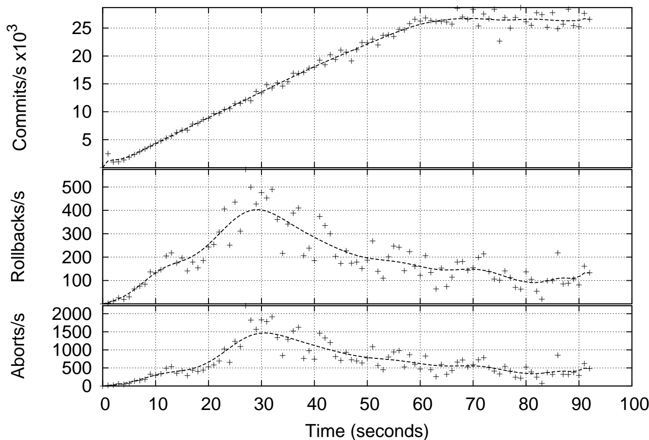


Figure 4. Commit, rollback, and abort rates for different workloads (the lines are an approximation for the data).

a local-area network in this case. For wide-area networks, the gap would be higher because the agreement time is a function on the communication delay. In addition, in case of failures the advantages of speculative events can be even more significant.

Also shown in Figure 5, the processing latency in the first replica, which emits both speculative and final events, is relatively stable. On the other hand, the processing latency in the second replica depends on the probability of speculations being successful and, thus, has a higher variability. Nevertheless, final messages from the second replica arrive sometimes earlier than final messages from replica one. Note that in a wide-area network the variability of communications would be higher and being able to also use final messages from the second replica would become advantageous.

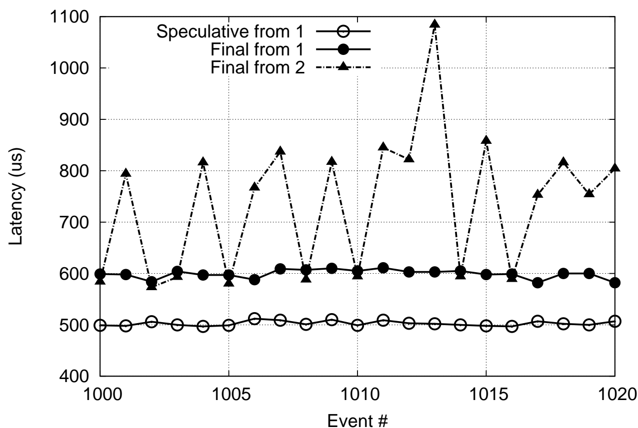


Figure 5. Individual processing latencies for a replicated component.

Finally, in our last experiment, we consider the case of failures. As discussed above, in the failure free case, having speculative events being forwarded earlier may help overlapping the agreement for ordering with computation and can even give some time advantage to the downstream operator. However, the main contribution of speculative events is when a failure occurs. This case is illustrated in Figure 6. In this execution a non-lowest id replica fails, i.e., the surviving replica is the one that has its speculative events considered by the downstream node. Observe that when the replica fails, the surviving replica will not be able to confirm the order of the messages and, consequently, will not be able to issue final events until it detects that the replica really failed. Nevertheless, the speculative events will still be generated and the computations downstream will be initiated early. In the case when the replica with the lowest id fails, the advantage of the speculative events will not be available as the downstream node would have to suspect the lowest id replica before it starts considering speculative events from another replica. It would be possible, however, to consider two failure detectors. One failure detectors is perfect and triggers the restart of the replica while the other, non-perfect but with a more aggressive detection time, can be used for non-critical decisions like changing the preferred origin for speculative events. We are currently investigating whether this two-failure detector approach can be useful in practice.

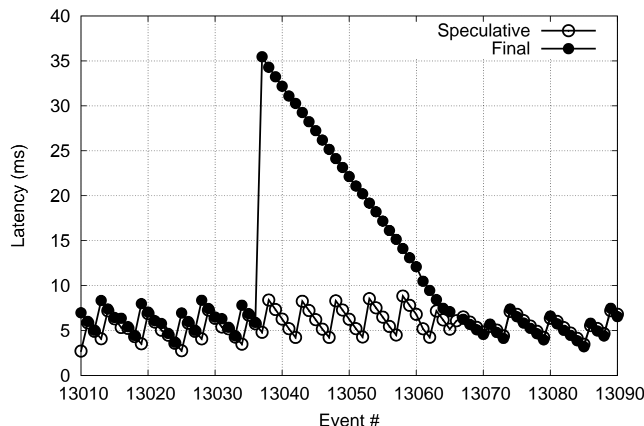


Figure 6. Effect of failure in the generation of final events.

V. Related Work

Active replication [1] and the state-machine replication approach [2] have been studied for general distributed systems for a long time. To address some of its limitations like the inability to handle non-determinism (multi-threading being one type of non-determinism), variations like semi-passive [5] and semi-active replication [6], [1] were proposed. In semi-active replication, one replica (the leader) takes the non-deterministic decisions and forwards these to the other replicas (the followers). Semi-active replication can be used to solve the problem of multi-threaded executions [6], but it requires collecting and communicating scheduling decision and may cause the followers to lag behind the leader. Similarly, in [19] the authors evaluate a

mechanism to transmit all lock-acquisition decisions from the leader node to the follower nodes. In semi-passive replication, all the relevant computation, together with any potentially non-deterministic scheduling decisions, are done by the primary. The primary then forwards state changes to the replicas.

Another approach is to make multi-threaded replicas deterministic. In [20], the authors present a deterministic scheduler that guarantees that multi-threaded replicas will execute deterministically. Nevertheless, only one thread can be active at a time. Thus, no parallelism is indeed exploited. In [19], the authors are able to preserve some of the parallelism by dividing the execution in rounds so that the order of the lock-acquisitions are decided deterministically at the beginning of each round. In this case, threads that try to acquire a lock will block until all other threads try to acquire any lock. Then, when all threads are blocked waiting for locks a scheduler grants them in a deterministic ordering. Although, this approach enables more parallelism to be exploited, there are still considerable limitations, specially if the processing of the different threads is unbalanced.

More recently, with the increase in the popularity and widespread usage of ESP systems, the issue of fault tolerance has also been addressed. Three examples of recent related work done in this area can be found in [8], [21] and [9]. In [8] the authors consider active replication through a process pair. They restrict the system to completely deterministic operators (and thus, no multi-threading) and impose that both replicas receive events from a single source. In [21], the authors discuss fault-tolerance approaches for stream systems and present solutions considering passive replication (referred to as passive standby in the paper) and semi-active replication (referred to as active standby). Finally, in [9], the authors acknowledge the problem of the high performance costs of implementing semi-active replication and propose modifications to the systems and operators to allow them to execute independently. In essence, they add well synchronized clocks to the sources, punctuation on the streams (to guarantee that after punctuation t, no event with a smaller timestamp will be sent by that source) and sorting of events (between punctuation) on the operators. In fact, this time-based approach can be seen as an implementation of atomic broadcast and, as expected, latency costs will be added as events cannot be processed as soon as they are received. The operator must wait for the punctuation and then sort the events so that the same ordering of events can be guaranteed among the replicas. Further, this approach does not address the use of multi-threaded operators. In our work, we use a traditional atomic broadcast (which could also be internally implemented according a time-based approach, if desired). We minimize latency costs by allowing operators to speculate while waiting for the final ordering and by supporting multi-threaded operators. As a complementary mechanism, in [13] we propose the usage of speculation with passive replication to hide the costs of logging and checkpointing states in stable storage.

Also related is the seminal work on virtual time [22], which considers the general case of speculation in distributed systems. In this work, the authors consider the case that instead of waiting for messages, nodes can simply continue computing in the hope that no message with a timestamp in the past will arrive. Then, in the case a node receives such a message, it rollbacks to a state before that message. A practical problem is that a rollback in a node will cause messages from the past to be sent and potentially trigger further rollbacks in other nodes. Thus, in a general distributed system it is crucial to coordinate globally which checkpoints must be kept to avoid cascading rollbacks. In our case, we use STM to efficiently keep fine-grained checkpoints, but in contrast to virtual time we do not face the problem of cascading rollbacks as ESP applications are acyclic directed graphs. Finally, in [23], the authors use an approach slightly similar to ours that relies on optimistic delivery to trigger early processing of transactions in distributed databases.

VI. Conclusions

In this paper, we have proposed new techniques to efficiently support active replication in fault-tolerant distributed event processing systems. By using an STM-based speculation mechanism, we allow nodes to optimistically start processing events before their final delivery (before their respective order is know with certainty). We have shown how the latency costs can be further decreased by also enabling one of the replicas to forward speculative events to downstream components even before final ordering is known. We have then addressed the case of multi-threaded actively replicated components. By using the transactional mechanisms provided by the STM, we can guarantee consistent executions among independent replicas. We have finally presented evaluation results that demonstrate the benefits and efficiency of our approach on a running system.

Acknowledgment

This research is conducted within the FP7 Integrated Approach to Transactional Memory on Multi-and Many-core Computers (VELOX) project (ICT-216852) supported by the European Commission.

References

[1] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active replication in delta-4," in *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, Jul 1992, pp. 28–37.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, 1990.

[3] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed systems (2nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216.

[5] X. Défago, A. Schiper, and N. Sergent, "Semi-passive replication." Los Alamitos, CA, USA: IEEE Computer Society, 1998, p. 43.

[6] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet, "Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive," in *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 90.

[7] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: high-throughput stream programming in java," *SIGPLAN Not.*, vol. 42, no. 10, pp. 211–228, 2007.

[8] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 827–838.

[9] J.-H. Hwang, U. Cetintemel, and S. Zdonik, "Fast and highly-available stream processing over wide area networks," in *ICDE 2008: Proceedings of the 24th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, April 2008, pp. 804–813.

[10] "Streammine," http://streammine.inf.tu-dresden.de, July 2009.

[11] C. Fetzer, A. Brito, R. Fach, and Z. Jerzak, "Streammine," in *Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies*, A. Hinze and A. Buchmann, Eds. Hershey, PA, US: IGI Global, 2009.

[12] N. Shavit and D. Touitou, "Software transactional memory," in *Symposium on Principles of Distributed Computing*, 1995, pp. 204–213. [Online]. Available: http://citeseer.ist.psu.edu/shavit95software.html

[13] A. Brito, C. Fetzer, and P. Felber, "Minimizing latency in fault-tolerant distributed stream processing systems," in *The 29th Int'l Conference on Distributed Computing Systems (ICDCS 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2009.

[14] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, "Speculative out-of-order event processing with software transaction memory," in *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. New York, NY, USA: ACM, 2008, pp. 265–275.

[15] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[16] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm, "Transactifying applications using an open compiler framework," in *TRANSACT*, August 2007.

[17] C. Fetzer, "Perfect failure detection in timed asynchronous systems," *IEEE Transactions of Computers*, vol. 52, pp. 99–112, Feb 2003.

[18] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[19] C. Basile, Z. Kalbarczyk, and R. K. Iyer, "Active replication of multithreaded applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 5, pp. 448–465, 2006.

[20] R. Jiménez-Peris, M. Patiño Martínez, and S. Arévalo, "Deterministic scheduling for transactional multithreaded replicas," in *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, 2000, pp. 164–173.

[21] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *ICDE 2005: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790.

[22] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[23] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 4, pp. 1018–1032, July-Aug. 2003.