Optimistic Database-Driven Distributed Real-Time Simulation (05F-SIW-031)

Marcus Brohede
Sten F. Andler
School of Humanities and Informatics
University of Skövde
P.O. Box 408, SE-541 28 Skövde, Sweden
{marcus.brohede, sten.f.andler}@his.se

Sang Hyuk Son
Department of Computer Science
University of Virginia
151 Engineer's Way, P.O. Box 400740, USA
son@cs.virginia.edu

Keywords:

Distributed real-time simulation, distributed real-time databases, fault tolerance, eventual consistency, active databases, time warp

ABSTRACT: *In this paper we present an optimistic synchronization protocol for distributed real-time simulations that uses a database as communication and storage mechanism. Each node in the simulation is also a database node and communication in the simulation is done by storing and reading to the database. The underlying replication protocol in the database then makes sure that all updates are propagated. The progress in the simulation is optimistic, i.e., each node tries to simulate as far ahead as possible without waiting for input from any other node. Since the simulations are said to be real-time we must guarantee that no events can be delivered too early nor too late. Also, recovery of a node must be done within predictable time due to the real-time constraints. Since all updates in the simulation are done through transactions we have a well-defined foundation for recovery and we show how the recovery can be done deterministically. For the simulation to function (and keep deadlines) during network partitions we allow local commits in the database. This requires that all data required on a specific node must be reachable from that node, i.e., no remote accesses should be needed. However, allowing local commits may introduce conflicting updates. These conflicts are detected and solved predictably.*

1. Introduction

Optimistic Concurrency control in parallel and distributed simulation differs from conservative concurrency control in that they speculate on the future instead of blocking. The key idea is that blocking guarantees to "waste" computational time by doing nothing whereas speculating execution only wastes when the speculation was incorrect and a recovery is needed (usually by rolling back the simulation). In this paper we show how an optimistic concurrency control protocol can be integrated into a distributed active real-time database (DARTDB).

Today, simulations either use checkpoints to recover from crashes or forward recovery (e.g., extrapolate a new position if a position update is lost) in order to minimize wasted computations. For checkpoint based distributed simulations this means to regularly create checkpoints and in case of failure restart all simulation nodes at the latest common checkpoint. This type of recovery can

be implemented in HLA simulations [2]. In [10], Lüthi and Berchtold show that this recovery can be improved for distributed simulations, by limiting the number of nodes that need to be restarted at these checkpoints. Simulations that use forward recovery (often used in DIS [3]) need to perform some compensating action when it is discovered that the recovery is incorrect (e.g., when position updates enters that show that our extrapolation is incorrect). Real-time simulations on the other hand have focused more on accurate timing behavior, for example to test a control software's ability to respond to some event (or event sequence). If timing errors (or other errors) occur during simulation the simulation cycle is often so short that a total restart of the simulation is the simplest remedy. Some real-time simulations (also called hybrid simulations [4]) interact with the environment (e.g., a human operator or some existing machine) and generate external actions. A problem with recovery in such systems is that external actions may not be possible to undo. For example, after drilling a hole, firing a missile, or deploying a first-aid kit in the real-world. Because of the external actions returning to a previous state may not be possible and simply stop or delay execution in order for a recovering simulation node to catch up is not possible. Hence, for long-running real-time simulations with external actions, current recover approaches are not sufficient. For this type of applications there could be a need to mask failures, or at least bound the recovery time and make sure that this time is short enough to recover before a new action is expected from the simulation.

As criticality of real-time simulations increases, bounded recovery time becomes an important issue. As mentioned above, this is particularly true for simulations with external actions. For example, where there are humans-in-the-loop or hardware-in-the-loop, or if simulations are used to make critical decisions where humans are affected. In these types of simulations, failures that require restart of simulations are associated with a costly penalty (hard essential deadlines). For example, in a battle simulation with simulated and real entities (soldiers, fighter planes, tanks, etc.) a restart would require all participating entities return to there starting points. For simulated entities this is no problem, i.e., they are easily reset to their starting coordinates (for example by reading some configuration file). However, real-world entities must, land, drive, or walk to their respective starting points. This behavior can, if simulations are large, be very time consuming and as a result be very costly. Another example where deadlines in simulations are hard essential is decision support systems where time requirements are stringent, since accurate

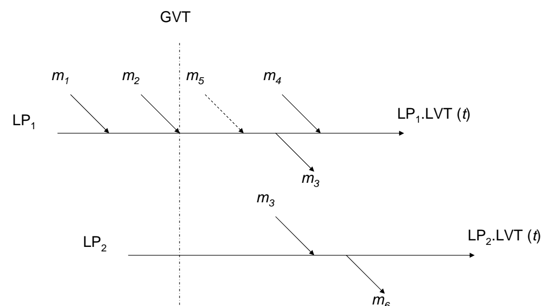

Figure 1. Rollback example in TW.

predictions based on simulated futures are key elements in sound decisions (see for example [7]). A late decision due to restarted (or rolled back) simulations can have severe consequences.

Continuously analysis of data from simulation often require an extra node tapping into the simulation to extract the data to some external storage facility (often a database). For example, in HLA this type of federates are called "passive listeners". In this paper we describe an architecture that use a database as infrastructure, i.e., no special attention is needed to gain access to the intermediate data of a simulation.

2. Related Work

Time Warp (TW) [8] is a common optimistic protocol used in discrete event simulations to guarantee that the dependency relation is kept throughout the entire simulation. TW allows independent logical processes (LP) to process events in their own event-lists as far ahead possible, i.e., until they need incoming events to process outgoing events, or until finished. However, if they receive an event with a timestamp t less then their current *local virtual time* (LVT) they must rollback their execution to this point t. A rollback effectively means to set the LVT to time t, reinstall the state just before this time t, and to "unsend" all messages sent during the time that has been rolled back with so called antimessages. Thereafter, the rolled backed node can start to re-execute all events from time t and forward. For the recovery old states as well as sent events must be kept. If no pruning of old states and

sent events are done the memory consumption would be uncontrolled. To handle this a *global virtual time* (GVT) defines a point behind which all states are stable, i.e., no recovery can go beyond the GVT. In Figure 1 all messages except m_5 arrive in numerical order. LP_1 will send m_3 and process m_4 before receiving m_5. The arrival of m_5 makes the speculative execution on LP_1 invalidated. The state on LP_1, therefore, must be brought back to just before the sending of m_3, which must be un-sent. Then the execution can start over at LP_1 with processing of m_5, re-sending m_3, and re-executing m_4.

Due to possible unbounded rollbacks TW is not suitable for real-time simulations. However, a restricted form of TW called No False Timestamps (NFT) Time Warp, was defined by [4] to provide a TW variant suitable for real-time simulations. NFT take in to account overhead such as state saving, state restoration, sending and receiving messages and antimessages, and can give an upper bound on the execution of a TW simulation given that no false events occur. A false event is an event that will be rolled back or canceled. If a simulation can be guaranteed despite it's rollback overhead R to meet all deadlines it is called *R-schedulable*. The R-schedule is generated by adding the overhead to all events in the simulation, i.e., the execution time of each individual event is increased by R. Unfortunately the class of simulations that can confirm to the requirements of NFT has been showed to be very limited and of little practical use [5].

In [5], Ghosh et al. show that optimistic simulation protocols that never sends incorrect messages (also known as aggressive no-risk simulations (ANR)) together with continuous generation of GVT provides a predictable way to execute optimistic real-time simulation. However, the continuous generation of GVT used in this protocol relies on nodes connected through a shared memory and it is not useable when nodes only are connected through a network. That is, this protocol is suitable for parallel systems, but not for distributed systems.

3. Database Approach

A DARTDB as defined in [1] provides a shared memory architecture that can guarantee local hard real-time requirements. By putting the data structures used in TW in the database, i.e., each LP's LVT, message queues, and state as well as the GVT, we can provide a database version of TW.

First, we define that each database node is also a database replica. The nodes are fully replicated, i.e., all information is available at all nodes. To ensure local real-time we define transactions to run locally and changes are propagated after commit to all other replicas. A replica is always locally consistent, but inconsistences between replicas can occur. The global state of the database, however, is said to converge to a globally consistent state if no more updating transactions enter the database. This variant of replication policy is called eventual consistency The replication (consisting of propagation and integration of updates) to other replicas can be bounded or unbounded depending on the network capability. For example, if the replicas are connected by a real-time network the replication can be bounded.

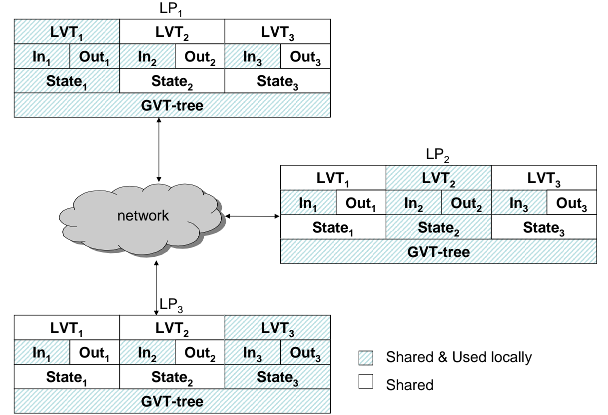

Figure 2. Data structures of TW in the database.

Second, each database node serves a LP. This means that each node will hold a local virtual time, and input and output queues for the corresponding LP. Shared between the LPs are the GVT, which is the lowest LVT among the LPs. Figure 2 illustrates a three node simulation. By using the tree structure defined in [5], where the LVTs of the LPs are leaf nodes and the GVT is the root node. The GVT can be calculated continuously and furthermore, since the database is active, we can specify a rule in the database to automatically update the GVT-tree. For example, the rule could look like this: ON $update(LP_i.LVT)$ IF $LP_i \neq root$ & $LP_i.LVT < LP_i.parent$ THEN $update(LVT_i.parent)$.

The basic operation of the database driven approach follows. Messages in input and output queues are tuples

consisting of a time of occurrence (timestamp) and the action(s), e.g., m_1(15,x=1) means that x is set to 1 at LVT 15. The queues are sorted on time of occurrence with the lowest time first. When the simulation starts each LP's LVT is set to ∞ and the GVT is set to 0. The processing on each LP_i consists of 4 steps: 1) take the first message (m_{head}) in LP_i's input queue and if the $m_{head}.timestamp$ is less then LVT_i then we have found a straggler and need to do recovery and start over processing from the recovery point, else set $LVT_i = m_{head}.timestamp$ and perform the actions in m_{head} on LP_i's state. 2) After successfully processing a message we must send the result to all LPs that use the result. This is done by writing the result in LP_i's output queue and the input queue of $\forall LP_j \in LP_j$ uses LP_i's result. 3) Update the GVT by checking if LVT_i is less then LVT_{parent} in the GVT-tree. 4) Check if $GVT = \infty$ and if true end the simulation.

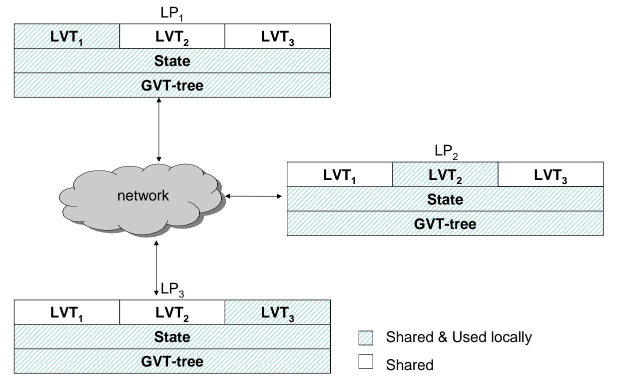

Figure 3. ANR TW in the database using active functionality.

4. Improved Database Approach

The database approach in section 3 does not use many of the features in the DARTDB, it merely uses the database as a message communication system and a storage facility. By adapting the ANR TW to the database an improved database approach can be obtained. For example, memory usage can be reducing by removing the message queues and rely more on the adaptive functionality.

Due to the fact that the database is fully replicated and active, we do not need to send messages between LP by storing values in the respective input queues. Updates can be done directly on the state variables, and active rules can monitor all these updates. The state variables them selves need to store old values up till the GVT. This means that for a simulation with n LP instead of $2 * n$ message queues and n states we could use 1 state and no message queues. The new state, however, would be a merge of the n states and each state variable would need to keep track of old values with $timestamps > GVT$. Figure 3 shows a tree LP that share state and have no input or output queues. The "messaging" is provided by the active functionality, which triggers the LP's to act when updates to state variables they use are detected.

In real-time systems all important tasks have deadlines and worst case execution times (wcet). For a task t with wcet t_{wcet} and deadline $t_{deadline}$ we say that the slack time for t is $t_{slack} = t_{deadline} - t_{wcet}$. Assuming that a task has a bounded recovery time $t_{recover}$ we can guarantee that the task t will finish before it's deadline iff $t_{slack} \geq t_{recover} + t_{wcet}$ (see figure 4). This is true under the assumption that a task fails at most once. Checkpoints can be used to avoid restarting a crashed task from the start. The use of checkpoints divides a task into smaller units that, once executed to completion, does not need to be reexecuted in case of a crash.

In figure 5, for example, the shaded part of task *t* does not have to be reexecuted even if a crash occurs in the later parts of the task. Instead the recovery kicks in and then the execution resumes at the checkpoint prior to the crash. The wcet for a task is then defined as $\sum_1^n wect_part_i$. If there is a crash in $part_j$ then the following formula must hold: $\sum_1^j wect_part_i + rt + \sum_j^n wect_part_i \leq wect + slack$. Factoring leads to $rt + wcet_part_j \leq slack$, i.e., the slack must be greater than the recovery time and the wcet for the crashed part.

A *recovery line* is a checkpoint taken at the same time in all participating nodes in a distributed system. In a distributed simulation, assume that we force a recovery line (just) before interaction with the real-world. Now, if any part of the simulation crashes it would rollback to the recovery line and then start to reexecute. If the next interaction with the real-world occur at time t_{next}, we must make sure that the recovery time $t_{recover}$ is less or else the entity in the real-world cannot rely on our simulation. For example, if a simulation starts a drill in the real-world. This would require the creation of a recovery line. Than if the simulation crashes and fail to recover in time. The

result could be that the drill produces a too deep hole. On the other hand if the detection and recovery can be guaranteed to be shorter than the next time to interact with the real-world, we can tolerate crashes in the simulation parts and still not drill too deep.

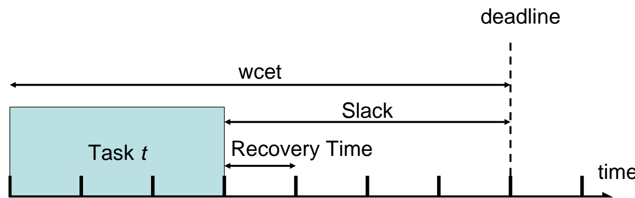

Figure 4. The slack time for a task *t*.

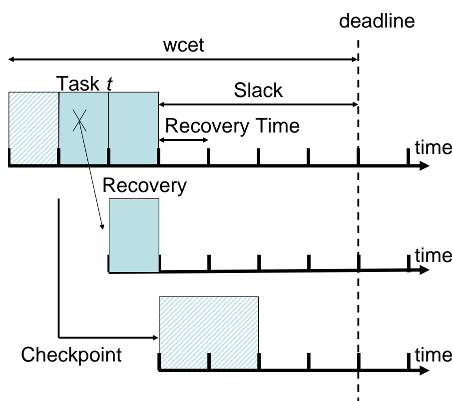

Figure 5. A task *t* using checkpoints.

5. Discussion

In this section we try to summarize and discuss the optimistic concurrency control protocol and the features of our database driven real-time simulation approach.

The major benefit in using our optimistic concurrency protocol is that real-time simulations with external actions can store all their state variable in a database and even though the simulation is distributed communication issues are hidden from the simulation engineers. Also, adding and removing nodes are potentially less cumbersome since all nodes are decoupled by the database, as opposed to for example distributed simulations with peer-to-peer connections.

As stated, the distributed database can be seen as white board or a shared memory. This simplifies the communication model for simulation engineers since, simulation nodes can communicating by writing and reading to this shared memory and all communication is catered for by the underlying replication mechanism. The database can give local real-time guarantees, since it allows local commits, is fully replicated and, main memory based. However, this design comes with a price: inconsistencies between nodes can exist even though locally at each node the database is consistent. To bound how long time replicas can be inconsistent due to conflicting updates a replication policy called eventually consistent replication is used. This policy detects and resolves conflicts in the distributed database in a timely and predictable way and it guarantees that the database converges to a globally consistent state [6].

Scaling is another issue that must be addressed when using this database approach. Full replication means that every node in the database hold it's own copy of the data, i.e., every LP's LVT, state, along with the GVT-tree would exist on every database node. This does not scale, i.e., the memory consumption growth is linear with respect to LPs (database nodes) and the number of replication messages growth exponentially. For example, with three nodes it would require three times the storage compared to a single-node database. To allow local commits, however, transactions cannot rely on any data outside

of the local node. This is why we need full replication. Transactions, on the other hand, only need the data they read or manipulate (write). This means that data, which never are accessed by transactions on a certain node, are not needed on that particular node for local commit reasons. By removing such superfluous data the important property of full replication, i.e., transactions never have to look elsewhere for data, is kept, but at a lower storage price. This feature is called virtual full replication [1]. One important point still remains though, data are replicated for fault-tolerance as well, and this must be considered when creating the database. One way to solve this issue is to segment the database. How to segment the database to achieve virtual full replication, as well as the expected decrease in storage requirements compared to regular full replication is ongoing research [11].

In addition, using a DARTDBS allows implicit storage of simulation data, i.e., no special attention is needed to save simulation states or results. This can be compared with, for example, HLA where one common way to achieve storage of simulation data is to have one passive federate tap into the federation and collect all simulation data [9].

For future work we intend to do a proof-of-concept implementation of the database driven optimistic real-time simulation and compare it to the existing shared-memory counterpart.

References

[1] S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS towards a distributed and active real-time database system. *ACM SIGMOD Record*, 25(1):38–40, March 1996.

[2] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly. The Department of Defense High Level Architecture. In *Proceedings of 1997 Winter Simulation Conference*, pages 142–149, Atalanta, GA, 7-10 December 1997.

[3] DIS Steering Committee. The DIS Vision, A Map to the future of distributed simulation. Technical Report IST-SP-94-01, Institute for Simulation and Training, Orlando, FL, May 1994. Version 1.

[4] K. Ghosh, R. M. Fujimoto, and K. Schwan. Time warp simulation in time constrained systems. In *Proceedings of the seventh workshop on Parallel and distributed simulation*, pages 163–166. ACM Press, 1993.

[5] K. Ghosh, K. Panesar, R. M. Fujimoto, and K. Schwan. Ports: A parallel, optimistic, real-time simulator. In *Parallel and Distributed Simulation (PADS'94)*. ACM, 1994.

[6] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *workshop on self-healing systems (WOSS'02)*, Charleston, SC, USA, November 2002.

[7] C. M. Harmonosky. Implementation issues using simulation for real-time scheduling, control, and monitoring. In O. Balci, R. P. Sadowski, and R. E. Nance, editors, *Winter Simulation Conference*, pages 595 – 598. ACM, 1990.

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

[9] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, 1999.

[10] J. Lüthi and C. Berchtold. Concepts for Dependable Distributed Discrete Event Simulation. In R. V. Landeghem, editor, *Proceedings of the Int. European Simulation Multi-Conference*, pages 59–66, 2000.

[11] G. Mathiason and S. F.Andler. Virtual full replication: Achieving scalability in distributed real-time main-memory systems. In *Proc. of the Work-in-Progress Session of the 15th Euromicro Conf. on Real-Time Systems*, pages 33–36, Porto, Portugal, July 2003.