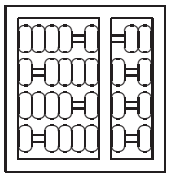

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

ChkSim: A Distributed Checkpointing Simulator

Gustavo M. D. Vieira *Luiz E. Buzato*

Technical Report - IC-05-034 - Relatório Técnico

December - 2005 - Dezembro

ChkSim: A Distributed Checkpointing Simulator

Gustavo M. D. Vieira* Luiz E. Buzato

Instituto de Computação, Unicamp
Campinas, SP, Brasil
{gdvieira,buzato}@ic.unicamp.br

Abstract

ChkSim is a portable tool to simulate the execution of checkpointing algorithms in distributed applications. It provides quantitative data to system and algorithm designers, enabling the comparative assessment of these algorithms. This report describes the ChkSim simulation model, its software architecture and user manual.

1 Introduction

Checkpoints are key states of the processes taking part of a distributed computation. Stored in persistent memory, those states provide an abstraction of the process execution. A global checkpoint is a set of local checkpoints, one for each process. Not all global checkpoints correspond to states that could have been observed in the application during a normal execution. The observable global checkpoints are called consistent global checkpoints. Using consistent global checkpoints we can observe the progress of the computation in a distributed debugging system [16] or recover part of a previous computation in case of failure [6], among several possible uses.

A simple way to guarantee the consistency of global checkpoints is to suspend the execution of the distributed application, blocking its progress while the processes synchronize and coordinate the global checkpoint. If this blocking behavior is eliminated by freeing the processes to take local checkpoints asynchronously, it may not be possible to build a consistent global checkpoint from the local checkpoints. This property was first observed in the context of fault recovery and is called *domino effect* [12].

Quasi-synchronous checkpointing [11] is a compromise between the synchronous and asynchronous local checkpointing. This approach leaves the processes free to choose their own checkpoints, but force them to take some extra checkpoints according to a checkpointing algorithm. In general, the decision to force a checkpoint is made using control information piggybacked in normal application messages. The checkpoints taken freely by the processes are called *basic checkpoints* and the checkpoints induced by the algorithm are called *forced checkpoints*. Many quasi-synchronous algorithms have been proposed [1, 2, 3, 4, 7, 8, 9, 13,

*Partially supported by CAPES under grant number 01P-15081/1997.

14, 15, 16] that exhibit different strategies to force checkpoints and produce checkpoint and communication patterns (CCP) with different properties. It is often very difficult to access the impact of any specific checkpoint algorithm on the basic computation solely based on the CCP properties provided.

ChkSim is a portable tool that enables the simulation of checkpointing algorithms in any basic computation, providing system designers with quantitative data that can used to access the impact of these algorithms. ChkSim can also be used with carefully crafted basic computations, stressing specific algorithm properties, being also a useful tool for algorithm designers. It is implemented in Java and can be run in any platform that has a Java Virtual Machine available. It was developed with two goals in mind: correctness and reproducibility. To attain correctness, the design of the tool was made simple and many aspects of its implementation, including the algorithms, are verified by a comprehensive test suite. Reproducibility is guaranteed by a completely deterministic simulation model. ChkSim has proved itself a valuable tool to implement and test new checkpointing algorithms in our research group.

This report describes the ChkSim simulation model, software architecture and user manual. Section 2 presents the computational and simulation model, crucial elements of the simulator architecture described in Section 3. Section 4 contains the ChkSim manual. Section 5 concludes the report by summarizing the key aspects of ChkSim.

2 Computational and Simulation Model

A *distributed application* is a set $\{p_0, p_1, \ldots, p_{n-1}\}$ of n sequential *processes* that cooperate to execute a user application. The processes of a distributed application are autonomous, do not share memory or a global clock, and communicate only through the exchange of messages over a *communication network*. The message exchange mechanism lets all pair of processes communicate directly and guarantees that messages are not corrupted, but does not impose bounds on communication delays and allows messages to be delivered in any order. Each process has its execution modeled as a finite sequence of events, where e_i^k represents the k-th event executed by process p_i, e_i^0 is the initial event of p_i. The events are classified as internal events and communication events. The communication events are *send* message or *receive* message, all other events are internal events. Each process maintains a set of local variables that forms its *state* and *checkpoints* are internal events that have had their associated states made persistent. A *global checkpoint* is a set of local checkpoints, one for each process.

A quasi-synchronous checkpointing algorithm works by reacting to events generated by the basic computation. More precisely it:

- Piggybacks control information to application messages at the moment they are sent;
- Processes piggybacked control information when messages are received, forcing checkpoints if necessary;
- Updates its data structures when the application takes a basic checkpoint.

Our simulation model closely implements this discrete mode of operation of the quasi-synchronous algorithms, abstracting the basic computation as a sequence of send, receive and internal events. Specifically, we are not directly interested in time related factors. Time for us is just logical time [10], and the event sequences considered are all consistent with causality. More formally, we define a *distributed computation* C to be an ordered, finite sequence $(e_1, e_2, e_3, \ldots, e_{|C|})$ of distinct events, where each event e_l represents an event e_i^k for some i and k, and the following condition holds:

$$\forall i, j \leq |C|, i \geq j \Rightarrow e_i \not\rightarrow e_j.$$

This means that the order of the sequence C respects the partial order created the causal precedence relation.

Clearly, timing can affect the behavior of quasi-synchronous algorithms. For example, variable message transmission delays can reorder the reception of messages. However, the correctness of these algorithms doesn't depend on timing constraints and, as long as all event sequences respect causality, they can reflect all possible message orderings. Thus, our simulation is based on a sequence of communication events (send, receive) and internal events, representing a distributed computation. These event sequences can be created freely as long as they don't violate causality. Moreover, the only internal events that are relevant to the operation of quasi-synchronous algorithms are the occurrences of basic checkpoints.

3 Software Architecture

Supporting our simulation model, ChkSim architecture is defined as a pipeline of events from a distributed computation. Events (send, receive and internal) are created in a *load generator*, then they are processed by a *simulator* and finally *run data* is extracted from the resulting stream of distributed computation events and checkpoints. Each stage of the pipeline is implemented by a software component with a very simple interface. Figure 1 shows ChkSim event pipeline.

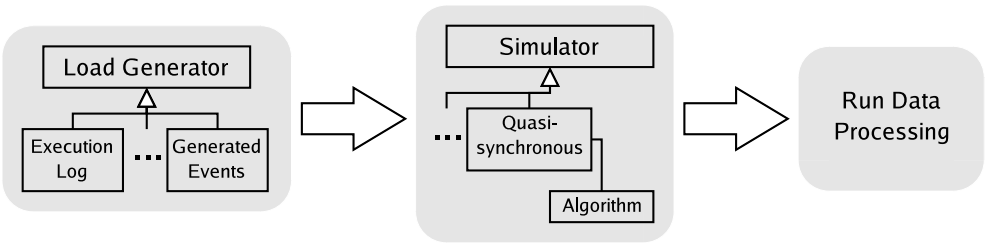

Figure 1: ChkSim event pipeline.

A load generator creates a sequence of events representing the distributed computation. For instance, a generated event sequence can reproduce an observed real computation from a properly total-ordered event log. Any causally consistent event sequence will do, thus

synthetic basic applications can be created by randomly generating events. The ChkSim package includes one such synthetic load generator, the statistic load generator.

The simulator processes the event sequence, realizing the events as it drives a checkpointing algorithm. If the event sequence represents the basic computation, the simulator acts as the communications network and checkpointing middleware. Communications events are created and dummy messages are instantiated and sent. The algorithm is notified of all events so it can piggyback control data in the messages and, if necessary, force checkpoints. Algorithms are implemented independently from the simulator and must respect an interface driven by these events. ChkSim implements a simulator for quasi-synchronous and non-blocking synchronous [5] checkpointing algorithms.

As the algorithm is interpreted by the simulator, its actions are recorded based on predefined performance metrics. This data is subsequently processed by the next stage of our pipeline. The data collection stage accumulates the data from these multiple simulation runs and performs some statistical analysis. The determinism of our simulation model guarantees that if all algorithms operate on the same event sequence, the results from these simulations runs are directly comparable.

4 ChkSim Manual

This section presents a manual on how to create and execute a simulation run. We start by describing how to create a load generator and how to use the provided statistical load generator. Then we show what defines a simulator and how to use and implement algorithms for the provided quasi-synchronous simulator. Finally we show how to put it all together, creating a complete simulation run.

ChkSim is free software and can be download from its project web page[1]. This page has instructions on how to download and configure the binary distribution. This is enough for most uses, but if you want to create new classes it may be helpful to also download the source package. In this manual we won't cover all aspects of the ChkSim programming interface. For a more detailed view of all classes, constructors and methods available please see the class reference documentation found in the project page.

4.1 Load Generator

The load generator is defined by a simple interface: `LoadGenerator`. Figure 2 shows the signature of this interface.

The method `getProcessNumber` should return the number of processes in the system. This is the only information regarding the system that the simulator needs. Other restrictions in the physical organization of the distributed system are enforced by the event stream only. The method `nextEvent` should return the next event of the sequence. Each event is represented by an object of the `Event` class. Finally, the method `reset` returns the load generator to its initial state. If the load generator is determinist, after each call to `reset` the stream of events produced by calling `nextEvent` should be the same.

[1] `http://www.ic.unicamp.br/~gdvieira/chksim/`

```
public interface LoadGenerator {
    int getProcessNumber();
    Event nextEvent();
    void reset();
}
```

Figure 2: `LoadGenerator` interface.

Any class that implements this interface can function as a load generator. It is easy to build load generators that recreate real computations using their execution log, for example, by simply reading the logs and piping the events through the `nextEvent` method. However, it is important to stress that the event sequences created by any load generator must be consistent with causality, so are any event log used to create them.

4.2 Statistic Load Generator

We have included in ChkSim a concrete implementation of a load generator that produces synthetic basic computations based on predefined distribution of events in the event sequence. This load generator is implemented in the class `StatisticLG`, that has a constructor with the following signature:

```
public StatisticLG(String network, String priorities, long seed)
```

The physical layout of the system is loaded from a network definition file and the event sequence is driven by a priorities definition file and a seed. Both network and priorities definitions *restrict* the computational model as a way to constrain the system to behave similarly to a target application. This statistic load generator is deterministic. Two statistic load generators created with the same seed will generate the same sequence of events.

The communication network is defined by a directed graph, where each vertex represents a process, and messages can only be sent from process p_i to process p_j if there is an edge connecting p_i to p_j. Generated communication events always reflect a message transversing those edges. This graph is defined in an XML file as shown in Figure 3. The definition is done inside the `network` tag, with nested `link` tags. Each link represents a directed communication channel from one process to the other. The `network` tag has two attributes: `size` defines the number of processes in the distributed system. The `type` attribute indicates if all links are to be defined individually or if all processes can communicate with all other processes. For the latter case, the definition of a complete network would be: `<network type="complete" size="6"/>`, with no links.

The priorities are defined by process and by event inside each process. These priorities define, in a manner unrelated to time, different orderings of events for event sequences. Figure 4 shows a XML file with a priority definition. The definition is done inside the `priorities` tag, with nested `process` tags. Each process is identified by its pid and has four priorities attributes: `priority`, `internal`, `send` and `receive`. The attribute `priority` defines the relative priority among processes. A process with priority "2" is two times more

```
<?xml version="1.0"?>
<network type="graph" size="6">
  <link from="0" to="1"/>
  <link from="0" to="2"/>
  <link from="1" to="2"/>
  <link from="2" to="4"/>
  <link from="3" to="0"/>
  <link from="3" to="5"/>
  <link from="4" to="3"/>
  <link from="5" to="1"/>
</network>
```

Figure 3: Network definition file.

likely to have events in the sequence than a process with priority "1". In a similar manner, but restricted to each individual process, the attributes `internal`, `send` and `receive` define the relative priority of each event for the specified process. For all processes not listed in the priorities definition file, the priority for all attributes is set to "1". If a process not present in the system is listed in the priorities definition file, its entry in ignored.

```
<?xml version="1.0"?>
<priorities>
 <process pid="0" priority="3" internal="3" send="2" receive="3"/>
 <process pid="1" priority="2" internal="1" send="2" receive="3"/>
 <process pid="3" priority="5" internal="3" send="2" receive="1"/>
 <process pid="8" priority="4" internal="1" send="2" receive="1"/>
</priorities>
```

Figure 4: Priorities definition file.

4.3 Simulator

Usually it is not necessary to define a new simulator, however it might be necessary to instantiate one of the defined simulators directly to perform statistic functions not implemented. Figure 5 shows the interface of the `Simulator` class.

The constructor of the class takes as arguments a load generator and the class name of an algorithm suitable for the type of simulator. The method `run` executes the simulation for a fixed number of communication events from the load generator, or it executes the simulation until there are no more events, if the zero arguments version is called. The method `getMetric` returns the value of a specified algorithm metric. Each type of simulator has its own set of metrics, identified by a metric name.

```
public abstract class Simulator {
    public Simulator(LoadGenerator lg, String algorithmClass);
    public abstract void run(int eventNumber);
    public void run();
    public abstract int getMetric(String metric);
}
```

Figure 5: `Simulator` class.

4.4 Quasi-Synchronous Checkpointing Simulator and Algorithms

One concrete instance of a simulator implemented in ChkSim is the quasi-synchronous simulator. This simulator is implemented in the `QuasiSynchronousSimulator` class, it uses algorithms of the `QuasiSynchronousAlgorithm` class and computes two execution metrics: "Basic" for the number of basic checkpoints and "Forced" for the number of forced checkpoints.

The quasi-synchronous simulator is instantiated and behaves exactly as the generic simulator described in the last section. It drives quasi-synchronous algorithms implemented by subclassing `QuasiSynchronousAlgorithm`. This abstract class performs some general setup operations and keeps track of performance metrics, the actual behavior of the algorithm should be implemented by concrete subclasses. Figure 6 shows the methods that need to be implemented for any concrete subclass of `QuasiSynchronousAlgorithm`.

```
public abstract class QuasiSynchronousAlgorithm {
    protected abstract void doInit();
    protected abstract void doTakeBasicCheckpoint();
    protected abstract Serializable doSendMessage(int destination);
    protected abstract void doReceiveMessage(int sender,
                                             Serializable message);
}
```

Figure 6: `QuasiSynchronousAlgorithm` abstract methods.

The `doInit` method should perform algorithm specific initializations and is called when a new instance is activated. The method `doTakeBasicCheckpoint` is called every time the basic computation takes a checkpoint, giving a chance for the algorithm to update its data structures. The method `doSendMessage` is called every time a message is to be sent and the algorithm should return the control data to be piggybacked. The method `doReceiveMessage` is called every time a message is received, giving a change for the algorithm to update its data structures and, if necessary, to force a checkpoint.

To implement these methods, the algorithm should use two concrete methods from the class `QuasiSynchronousAlgorithm`: `takeBasicCheckpoint` and `takeForcedCheckpoint`. These methods represent how the algorithm affects the basic computations and should

be called so that `QuasiSynchronousAlgorithm` can update its data structures and metric counts. An example of the implementation of the BCS algorithm [4] is in Figure 7.

```
public class BCS extends QuasiSynchronousAlgorithm {
    private int lc;
    protected void doInit() {
        lc = 0; takeBasicCheckpoint();
    }
    protected void doTakeBasicCheckpoint() {
        lc++;
    }
    protected Serializable doSendMessage(int destination) {
        return new Integer(lc);
    }
    protected void doReceiveMessage(int sender,
                                    Serializable message) {
        int messageLC = ((Integer) message).intValue();
        if (messageLC > lc) {
            takeForcedCheckpoint(); lc = messageLC;
        }
    }
}
```

Figure 7: The BCS algorithm.

4.5 Simulation Runner

A simulator, using an event sequence, drives a single checkpointing algorithm. To create a complete simulation run where several algorithms are compared using different event sequences, we use a simulation runner. The runner is implemented in the `Runner` class and is configured by a run definition file. Usually it is not necessary to interact directly with this class, as the ChkSim wrapper class can be used to start a simulation (see Section 4.6). However, accessing this class directly may be useful for computing statistical information not available through the ChkSim wrapper.

The constructor of the `Runner` class takes as argument a run definition file. Figure 8 shows and example of this file. The main tag is `run`, and it has the `events`, `iterations`, `seed` and `simulator` attributes. The `events` attribute define how many communications events per process will compose the basic computation, giving its overall duration. The number of events specified in this tag is only a per process average, some processes may end up executing more or less communication events than the number indicated here. The `iterations` attribute indicates how many times each data point will be run, the result value for each metric computed as the average of each iteration. The `seed` attribute defines the seed to be used, if the load generator requires a seed. Finally, the `simulator` attribute

indicates the class of the simulator to be used.

```
<?xml version="1.0"?>
<run events="100" iterations="5" seed="42"
     simulator="org.sagui.chksim.QuasiSynchronousSimulator">
 <metrics>
  <metric name="Basic"/>
  <metric name="Forced"/>
 </metrics>
 <algorithms>
  <algorithm class="org.sagui.chksim.algorithm.qs.BCS"/>
  <algorithm class="org.sagui.chksim.algorithm.qs.FDAS"/>
 </algorithms>
 <datapoints>
  <datapoint label="2" loadgenerator="org.sagui.chksim.StatisticLG"
             seed="true" args="complete-2.xml, priorities.xml"/>
  <datapoint label="3" loadgenerator="org.sagui.chksim.StatisticLG"
             seed="true" args="complete-3.xml, priorities.xml"/>
  <datapoint label="4" loadgenerator="org.sagui.chksim.StatisticLG"
             seed="true" args="complete-4.xml, priorities.xml"/>
 </datapoints>
</run>
```

Figure 8: A simulation run definition file.

The run is defined by the `metrics`, `algorithms` and `datapoints` tags. The `metrics` tag contains a `metric` tag for each metric to be computed, identified by its name, and provided by the selected simulator. Similarly, the `algorithms` tag contains an `algorithm` tag for each algorithm to be simulated, identified by its class name. All algorithm must be of a type compatible with the simulator being used and the class implementing the algorithm should be visible to the simulation runner. The `datapoints` tag holds a `datapoint` tag for each data point to be collected. A single data point defines an event sequence by its load generator using the attributes `label`, `loadgenerator`, `seed` and `args`. The `label` attribute provides an identifying label for this data point. The `loadgenerator` attribute contains the class name of the load generator to be used and the `args` attribute contains a comma separated list of string arguments to be passed to the load generator constructor. The class of the load generator should be visible to the simulation runner. Finally, the `seed` attribute indicates if the load generator needs to receive a seed for its pseudo-random number generator.

Once created, the most important methods of a `Runner` object are `run` and `getRunData`. The method `run` starts the simulation run, according to the parameters specified in the run definition file, collecting the data for all listed metrics. Once the run is complete, the method `getRunData` can be used to get access to all data collected, stored in a object of class `RunData`. This class has methods to access all metrics by name and to get the raw

values computed, averages and standard deviation for all data points and algorithms.

4.6 The ChkSim Wrapper

Putting all together, the ChkSim wrapper loads a run definition file, runs the simulations and creates GnuPlot[2] files with charts for all selected metrics. This wrapper is implemented in the `ChkSim` class and is an example of the type of pipeline building applications we can build using the `Runner` class as engine.

To start a simulation using the wrapper, first download and configure ChkSim as described in the project page. Then, download or create a simulation run definition file, for example: `run.xml`. Fire the simulator with:

```
$ java -cp chksim.jar:jdom.jar org.sagui.chksim.ChkSim run.xml
```

After the simulation has finished, two files for each metric will be created on the current directory: `run-<metric>.data` and `run-<metric>.plot`. Actually, the "run" part of the file name is derived from the run definition file name, so the more general file name would be: `<run-file>-<metric>.data` and `<run-file>-<metric>.plot`. The first file contains the averages per data point for each algorithm in a format suitable for the GnuPlot program. The second file is a GnuPlot command file that displays a chart with the results on the screen. For example, to see the chart for the "Forced" metric, use:

```
$ gnuplot run-Forced.plot -
```

5 Conclusion

In this report we presented the ChkSim distributed checkpointing simulator, described its software architecture and user manual. When designing this simulator our objective was to create a valuable tool for system and algorithm designers. This motivated us to build it in a way that would make easy to create and reproduce simulation runs. Thus, ChkSim is free software, freely available with complete, documented source code. Simulation runs created with the tool can be reproduced and audited with minimum effort. We invite the interested reader to visit the project page[3], download and try the tool.

References

[1] Roberto Baldoni, Jean-Michel Hélary, Achour Mostefaoui, and Michel Raynal. A communication-induced checkpoint protocol that ensures rollback dependency trackability. In *27th IEEE Symposium on Fault Tolerant Computing (FTCS)*, pages 68–77, June 1997.

[2] `http://www.gnuplot.info/`

[3] `http://www.ic.unicamp.br/~gdvieira/chksim/`

[2] Roberto Baldoni, Francesco Quaglia, and Bruno Ciciani. A VP-accordant checkpointing protocol preventing useless checkpoints. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 61–67, 1998.

[3] Roberto Baldoni, Francesco Quaglia, and Paolo Fornara. An index-based checkpoint algorithm for autonomous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):181–192, February 1999.

[4] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, October 1984.

[5] Guohong Cao and Mukesh Singhal. Checkpointing with mutable checkpoints. *Theor. Comput. Sci.*, 290(2):1127–1148, 2003.

[6] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Yi-Min Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996.

[7] Islene Calciolari Garcia and Luiz Eduardo Buzato. An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS '04)*, pages 126–135, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Islene Calciolari Garcia, Gustavo Maciel Dias Vieira, and Luiz Eduardo Buzato. RDT-Partner: An efficient checkpointing protocol that enforces rollback-dependency trackability. In *19º Simpósio Brasileiro de Redes de Computadores (SBRC)*, May 2001.

[9] Jean-Michel Hélary, Achour Mostefaoui, Robert Netzer, and Michel Raynal. Preventing useless checkpoints in distributed computations. In *16th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 183–190, October 1997.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[12] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[13] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[14] K. Venkatesh, T. Radhakrishnan, and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–303, July 1987.

[15] Gustavo Maciel Dias Vieira, Islene Calciolari Garcia, and Luiz Eduardo Buzato. Systematic analysis of index-based checkpointing algorithms using simulation. In *IX Brazilian Symposium on Fault-Tolerant Computing (SCTF)*, pages 31–42, March 2001.

[16] Yi-Min Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.