Chapter 6

TPC-C based Dependability Benchmark Focusing on Hardware Faults

Abstract

Large transactional systems are usually at the very centre of the IT infrastructure of companies. Even short downtimes of such a system are very expensive. Unexpected losses of data may even put a company out of business. To be able to evaluate the dependability of transactional systems is, therefore, of great importance. This chapter presents TPC-C-Depend, a dependability benchmark for online transaction processing (OLTP) systems. TPC-C-Depend is an extension of TPC-C and closely follows the form and structure of the latter. Like TPC-C, TPC-C-Depend, is specification-based and divided into a number of clauses. These describe the class of on-line transaction processing systems to which the TPC-C-Depend applies, present the dependability measures evaluated, lay down a set of implementation guidelines and define all important terms. The benchmark target is the data base management system.

The two measures provided by TPC-C-Depend are the stationary system availability and the total cost of failures. They area evaluated by combining measures obtained from experimentation on the target system (e.g., the percentages of the various failure modes) and information from outside the benchmark experimentation (e.g., the failure rate, the repair rate and the cost of each failure mode).

6.1 Introduction

Performance benchmarks for transactional systems have been around for a long time. The de facto standards that are accepted and supported by database, operating system and hardware vendors alike are the benchmarks from the Transaction Processing Performance Council (TPC). There are a number of different TPC benchmarks for different types of database systems. The current performance benchmark for large databases in a client-server or three-tier environment is the TPC-C [TPC-C 2002]. We have therefore based our research on the TPC-C. The TPC-C is a document, specifying how the benchmark must be implemented. The TPC ensures conformance with the specification before the benchmarks results may be published. Implementation details must be laid open in a full disclosure report, so that independent parties can reproduce the benchmark.

In this chapter, we define a benchmark for transactional systems, based on TPC-C, referred to as TPC-C-Depend. TPC-C-Depend is an extension of TPC-C and closely follows the form and structure of the latter. The TPC-C is divided into a number of clauses. These lay down a set of implementation guidelines, define any important terms, describe the class of on-line transaction processing (OLTP) systems to which the TPC-C applies and the result measures obtained.

The chapter is composed of six sections and an Annex. Section 6.2 gives an overview of the approach used to build a dependability benchmark for transactional systems and defines the benchmark measures to be evaluated by TPC-C-Depend. Section 6.3 presents the specifications of TPC-C-Depend (i.e., the documents describing the workload, the general requirements a system under benchmarking must fulfil, and the rules concerning disclosure of the benchmark results). Section 6.4 describes an implementation example of TPC-C-Depend as well as example of results obtained using the implementation. Section 6.5 details how the properties of this dependability benchmark, such as portability, scalability or reproducibility were tested. Section 6.6 concludes the chapter.

Because full disclosure reports are essential to make any benchmark result reproducible, trustworthy and transparent, we devote Annex 6-A to a discussion of formal full disclosure reports to ensure reproducibility. The Annex details the specification of TPC-C-Depend.

6.2 Building Dependability Benchmarks for Transactional Systems

The aim of this section is to give an overview of the approach used to build the dependability benchmark for OLTP systems, presented in this chapter. Section 6.2.1 outlines our benchmarking approach to building a dependability benchmark for transactional systems. It refines the general concepts presented in Chapter 1 and defines the set of measures to be evaluated by TPC-C-Depend Section 6.2.2 defines the experimental and final evaluated dependability measures and how to obtain them. Section 6.2.3 defines the cost of failure measure. Section 6.2.4 summarises the measures that should be defined clearly in the full disclosure report.

6.2.1 Benchmarking Approach

Figure 6.1 illustrates the interrelations of the different components involved in dependability benchmarks. It is important to note, that the part *above* the benchmarking interface is defined in the specification of the dependability benchmark. This part is fixed by definition and is defined in a portable and general manner. The part *below* the benchmark interface cannot be defined in the benchmark specification, since, in general, the system under benchmarking is not chosen before an actual benchmarking experiment is to be performed. This part is usually different for each benchmarking experiment and includes, for example, the actual hardware and software used in this specific benchmarking implementation. As can be seen from Figure 6.1, the faultload and background load may be part of the specification or the implementation. It depends on the kind of faultload or background load chosen. Contrary to generic failure modes, a hardware faultload can only be determined during the implementation, because which hardware faults can occur is strongly dependent on the exact hardware chosen and cannot, therefore, be defined at a time, when the hardware used is not yet known.

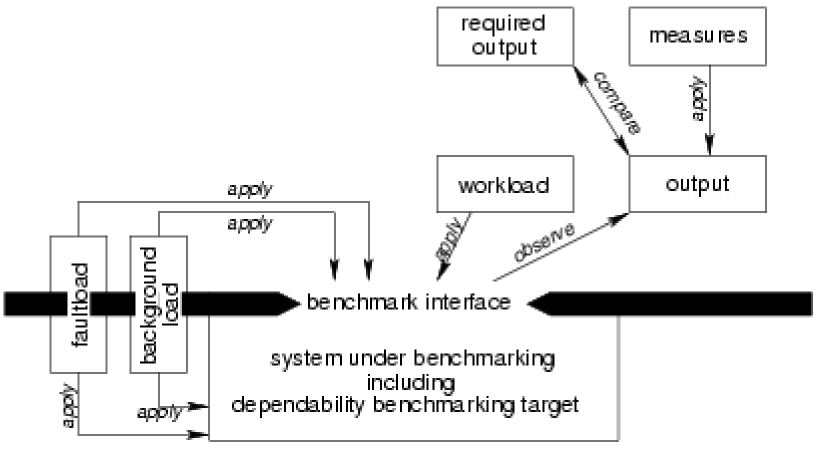

Figure 6.1: Interrelationships of Dependability Benchmarking Elements

A practical approach to creating a dependability benchmark specification is to take an existing performance benchmark and extend it into the dependability domain. Figure 6.2 sketches how the dependability extensions are integrated with the original performance benchmark. The question now is what aspects must be addressed when extending a performance benchmark into a dependability benchmark? It is important here to remember, that we are talking about the benchmark *specification*, for example, the dependability equivalent to the TPC-C or SPECWeb99 documents. How to *implement* a benchmark according to this specification is not the focus on this chapter, although rules concerning the implementation are given where the way of implementation directly affects measures and results.

Dependability benchmarks compare the dependability of alternative or competitive systems via specific interfaces. These interfaces have to be described in the benchmark document.

Contrary to pure performance benchmarks, a dependability benchmark requires also specification of the correct workload output in order to be able to decide whether or not the system under test performed correctly in the presence of faults. The generated output must be compared to the predefined expected output.

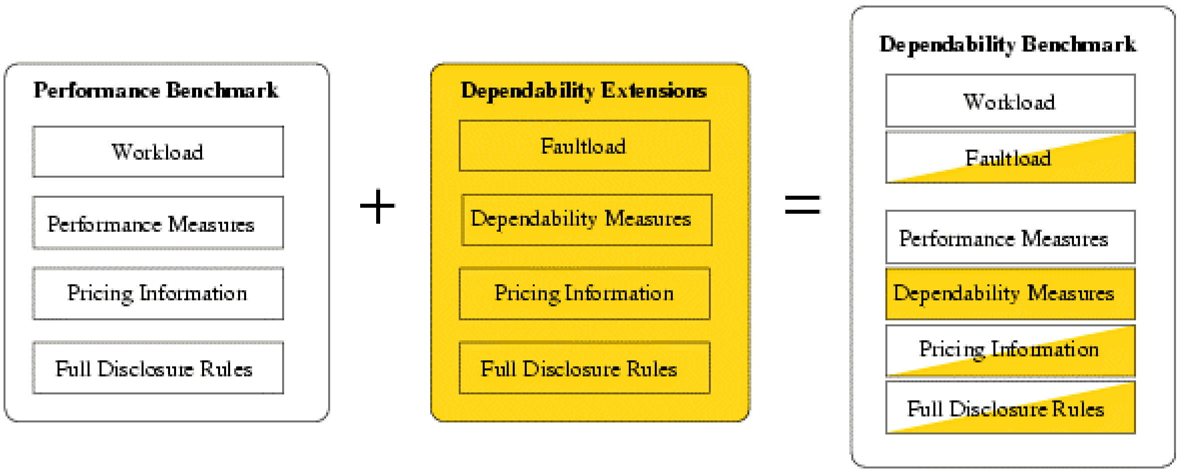

Figure 6.2: Deriving Dependability Benchmarks from Well-Known Performance Benchmarks

The workload does not need to be changed; the faultload on the other hand is a new dimension. When faults occurring in the environment of the benchmarking target may cause it to fail, using performance measures alone to characterize the system may result in a skewed view. We therefore add dependability measures. To calculate these measures we might also need new methods to observe the system behaviour.

Some performance benchmarks contain pricing information covering several years of service of the system under benchmarking and already include price information on maintenance actions. Even though, pricing information must be extended to cover financial loss in case of system failure as well as repair actions (which may be outside of routine maintenance).

The rules governing the contents and form of the full disclosure report must also be extended to include the new aspects dependability introduces. The major new aspect here is the fact, that a non-deterministic element is introduced.

The selection of a faultload for a given application domain should be governed by similar rules as the selection of a workload. In a performance benchmark, the system is assumed to be functioning perfectly and fault free. The faultload introduces an element of surprise, which is absent from performance benchmarks. When applying a faultload, the behaviour of the system may become quite unpredictable and non-deterministic. Even the same fault occurring at different times may lead to different system behaviour.

Of course, the true matter of interest is the system behaviour after a fault has occurred, not the fault itself. Hence, it is important to classify the system behaviour into *failure modes* (FM). A FM is a subset of possible observable behaviours of the SUB. Failure modes partition the full behaviour of the SUB. i.e., the failure modes should not overlap and the union of all failure modes must be the set of all possible behaviours.

Our dependability benchmark includes a list of FM considered relevant to the application domain. A performance benchmark has only a single mode "*fully functional*"; a dependability benchmark must have at least two modes *fully functional* and *unknown*. All system behaviour that cannot be classified is put into the *unknown* FM. When calculating costs of severity of FM, the *unknown* FM is considered as identical to the most expensive or most severe FM. In this way, all estimations and results we calculate will never be better than their true value. Otherwise, the specification should not contain a ranking of the severity of the FMs, as this would associate some idea of a cost with an FM. The cost of a FM just like the cost of the hardware used in the benchmark cannot be part of the specification. It is part of the benchmark implementation and must be laid open in the full disclosure report.

Different types of faults may occur in a system. Hardware Faults include all types of faults in the hardware. Apart from general rules and guidelines on the selection of hardware faults information on hardware faults cannot be included in the dependability benchmark specification. This information (such as assumed mean time between failures of hardware components) is obtained during benchmark implementation and must be included in the full disclosure report.

Hardware faults introduce probability into the picture. Therefore, a single run of the workload is no longer adequate, as is the case for pure performance benchmarks. The team conducting the benchmark should comply with the following rules concerning hardware faults: The number of experiments conducted for each hardware fault must be relative to the value of the rate of occurrence of this fault.

When selecting faults for inclusion in the faultload of the dependability benchmark it is important to focus on *relevant* faults. We define relevant faults to be those with a large rate of occurrence in the system under benchmarking.

Remark: In our experiments we also considered faults made by human operators of the system. Of course, operator faults only apply to systems that are routinely maintained and operated by humans, such as databases or web servers, but probably not embedded systems. Contrary to hardware and software faults, operator faults for a given application domain can be defined in the specification. (If we ask "Do operator faults occur and what happens then?", the frequency of occurrence of operator faults can, of course, not be determined in the specification.)

6.2.2 Dependability Measures

Performance benchmarks run the workload once for a specified period of time. All measures of the performance benchmark are derived from this single run. For a dependability benchmark more than a single run with the same fault injected is needed to derive meaningful dependability measures. This is due to the fact that a fault will often not always lead to the same FM with certainty but instead lead to several different FM with certain relative frequency for each. Since this frequency is a random variable, a good approximate value for this frequency can only be calculated when a large number of single runs are performed.

This section presents some guidelines for deriving benchmark measures from experimentation and defines dependability measures either obtained experimentally of calculated from our benchmark experiments.

6.2.2.1 Implementation Guidelines For Deriving Benchmark Measures

A dependability benchmark is conducted in three phases, shown schematically in Figure 6.3. We assume, at this point, that all the preparations, such as implementing the necessary benchmarking code, buying the necessary software and hardware, have already been completed. That is, before the set-up phase begins, all items necessary to implement and run the benchmark are available.

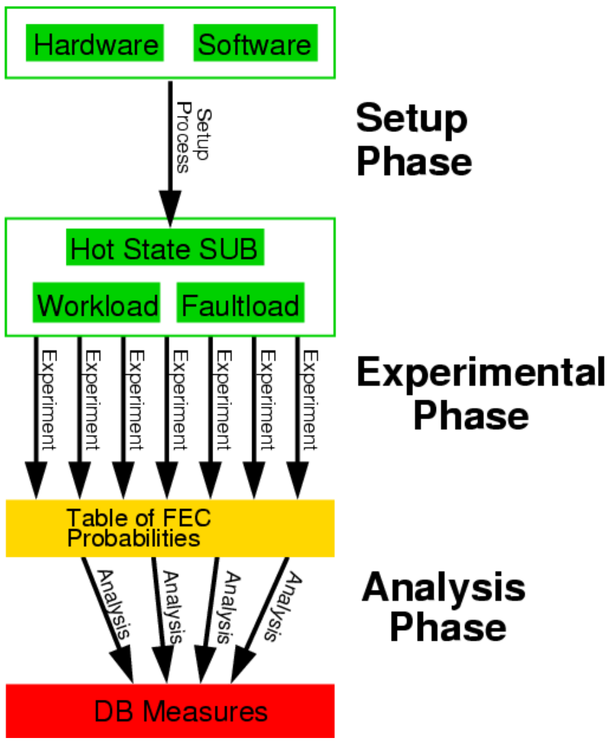

Figure 6.3: Phases of a Dependability Benchmark

The benchmark specification is an input to all phases and is therefore not shown in Figure 6.3. The set-up phase includes assembling the hardware and installing the operating systems and any application or benchmarking software either manually or automatically. The "hot state system under benchmarking" is the output of the set-up phase and one of the inputs of the experimental phase. Other inputs needed for the experimental phase apart from the hot state system under benchmarking are the workload, background load and faultload.

The term *experiment* refers to a single benchmark execution, with or without faultload.

A *Golden Run* derives measures without any faultload. The Golden Run is, in general, equivalent to execution of the performance benchmark without the dependability extensions. The Golden Run serves as reference for the other runs with activated faultload. All experiments apart from the Golden Run determine the effect of one single fault.

During the experimental phase a number of experiments with the following properties must be conducted:

- There must be at least one experiment without faultload. This is called the *Golden Run*.
- An experiment with faultload gathers information used to calculate the relative frequency of a certain failure mode with a certain confidence interval. The behaviour of the system under benchmarking observed during the experiment must be classified into a failure mode.
- The number of experiments must be high enough to derive appropriate confidence intervals.
- At the start of each experiment, the system under benchmarking must be in the same state. If the performance benchmark specifies that measuring is not begun until the system has reached its stable state, then we require also for a dependability benchmark derived from this performance benchmark that the faultload is not applied before the system under benchmarking reaches the stable state.

Performance benchmarks often require that the system under benchmarking reaches a stable state before the measuring is begun [SPEC 2000], [TPC-C 2002], i.e. the measures are taken while the system is running and providing services. Measures are not taken during initial start-up of the system.

During each experiment with a faultload the following observations should be recorded:

> **Fault** The fault applied in to the system under benchmarking in this experiment is recorded.
>
> **Failure Mode** The fault effect is observed and sorted into one of the failure modes of the dependability benchmark. If the set of failure modes includes a degradation of performance, the performance measures have to be recorded.
>
> **Performance Measures** The performance measures as described in the underlying performance benchmark are recorded.

The last phase is the analysis phase, during which the relevant measures are calculated from the data gathered during the experimental phase.

The degree of reliance must be included in the full disclosure report. The confidence intervals can be tightened by performing more experiments. A comparison of two different benchmarked systems is only valid if the confidence intervals of any measure do not overlap. Otherwise the statement of quality is hidden in statistical noise.

6.2.2.2 Experimental Measures: Failure Modes

As an intermediate result, the data collected during the experimental phase is gathered in the failure mode Table (The format of this table is shown in Table 6.5.). The column-headings are the failure modes FM_j. The row-headings are the single faults f_i that constitute the faultload.

Each cell in the table contains the relative frequency h_{ij} with which failure mode FM_j was observed under the condition that fault f_i occurred. h_{ij} is calculated as percentage, the number of times failure mode FM_j was observed when fault f_i was applied divided by the total number of times n_i fault f_i was applied. The relative frequencies h_{ij} are measured quantities derived during the experimental phase. The h_{ij} can therefore be seen as random variables. During each experiment with an applied fault f_i the system exhibits a behaviour that is classified as falling into a certain failure mode FM_j.

To fill the table of failure modes (TFM), information about the rates of occurrence of the faults included in the faultload or the costs of failure modes is not necessary. The table of failure modes contains data that — together with the fault rates and cost distributions — can be used to calculate dependability measures and confidence intervals.

6.2.2.3 Availability Measure

Given additional information about the faults f_i, it is possible to calculate the rates of occurrence of each failure mode FM_j, the repair rate of each failure mode FM_j or the probability of being in failure mode FM_j at some point in time.

Let $\rho = \sum_i r_i$ be the sum of all fault rates r_i, of the failure modes i, and n be the total number of experiments. Then we can calculate the number of experiments n_i in which we inject fault f_i as $n_i = n \cdot \frac{r_i}{\rho}$.

Let R_j be the rate of occurrence of a failure mode FM_j. R_j can be calculated from the information in the FMT if the fault rates r_i are known for all faults part of the faultload:

$$R_j = \sum_{i=0}^{N} \left(r_i \cdot hi_j \right) \qquad (6.1)$$

Because we should treat the relative frequencies h_{ij} as random variables, we should calculate a confidence interval for R_j.

Let Q_j be the repair rate of a failure mode FM_j. Q_j can be calculated from the information in the FMT, if the repair rates q_i are known for all faults of the faultload. The repair rate can be calculated from the mean time to repair (MTTR) as $q_i = 1/MTTR$. The equations are similar to those for R_j, but the repair rates q_i are substituted for the fault rates r_i.

$$Q_j = \sum_{i=0}^{N} \left(q_i . h_{ij} \right) \qquad (6.2)$$

Again, because the h_{ij} are random variables, we should calculate a confidence interval for Q_j.

Availability is a measure for the ability of a system to be functional at a certain point in time. It is a measure used for repairable systems.

The current availability $A(t)$ depends on variables such as the knowledge of the repair technicians, logistic support, repairability of the system. These variables are usually hard to quantify, which makes $A(t)$ very hard to calculate.

The stationary availability A is the limit of $A(t)$ for $t \to \infty$. Assuming that a system provides continuous service and is repaired when a failure is detected, we can calculate A as

$$A = \frac{T_{up}}{T_{total}} \qquad (6.3)$$

T_{up} is the uptime of the system, during which it is available. T_{total} is the total time the system is observed. These two measures can be calculated from the failure mode table, once the failure modes have been divided into two sets. Every failure mode must be in exactly one set. One set contains all those failure modes FM_M corresponding to unavailability of the system, S_U, the other all those representing a state FM_O, where the system is deemed available, S_A .

$$FM_O \in S_A \qquad (6.4)$$

$$FM_M \in S_U \qquad (6.5)$$

The uptime of the system can be calculated from the *mean time between occurrences* (MTBOj) of the failure mode FM j: $MTBO_j = 1/R_j$.

The total time is the sum of the uptime and the downtime of the system. The downtime can be calculated from the MTTRj of the failure modes: $MTTR_j = 1/Q_j$.

The availability A of the system under benchmarking can now be calculated as

$$A = \frac{\sum_{FM_j \in S_A} MTBO_j}{\sum_{FM_j \in S_A \cup S_U} MTBO_j + MTTR_j} \qquad (6.6)$$

Due to the fact that these are measures derived from random variables, we calculate minimum and maximum values for the availability and unavailability.

6.2.3 Cost Measure

For a fair comparison of systems performance measures are not enough. There is no doubt, for example, that a high-performance multiprocessor machine achieves better marks in a performance benchmark than a standard single-processor PC compatible. On the other hand, the price for the high-end machine's hardware alone is probably a multiple of that for the standard PC. Therefore, to ensure a fair comparison, the cost for providing the performance must be known, and those benchmark results meant for comparison should therefore include cost.

As a general guideline we can say, that what is benchmarked should be priced and what is priced should also be part of the system under benchmarking. This includes hardware and software costs as well as maintenance and failure costs.

The pricing of hardware, software and routine maintenance should already be part of the performance benchmark on which the dependability benchmark is based. The failure costs, specifically the fault and failure mode costs, are new in the dependability benchmark.

Costs are associated with each fault f and failure mode FM. The cost c_i associated with a fault f_i is the cost necessary to repair the fault. In the case of hardware faults, this may be the cost necessary to replace the faulty piece of hardware, including such things as the actual hardware cost and wages for the technician doing the repair job. For an operator fault, the cost could include wages for a senior operator who can undo the faulty action.

There is a hidden cost, which has so far been ignored, because we know the fault applied and observe the failure mode caused. In real life, the fault that caused a certain failure mode is not known in advance. Instead, it must be detected starting with the information known about the failure mode. Figuring out the exact fault will take a certain amount of time, depending on failure and fault information gathered from the system and of course the expertise of the human on the job. Some software may have useful information in log- or error files, some hardware may have status-LEDs (common for hard-disks in RAIDs, for example) which may make it easy to detect a fault, other systems may not be so forthcoming. We therefore suggest to fill a second table with the cost value needed to detect a fault, given a certain failure mode.

The row-headings of this table are the same failure modes; the column-headings are the faults. The values in the cells represent the cost d_{ji} needed to find out, that failure mode FM_j is indeed caused by fault f_i. There is also a cost C_j associated which each failure mode FM_j. If the failure mode entails loss of data, C_j could include the cost to restore and older version of the lost data from a previous backup as well as the loss of customers due to the loss of data. If a failure mode is a reduced performance mode, the cost would be due to the loss of online business, because of the sluggish response of the server to online requests.

The total cost X_j of failure mode FM_j is the cost from being in this failure mode added to the cost of getting out of this failure mode (repairing the system):

$$X_j = C_j + \sum_{i=0}^{N} (c_i + d_{ij}) \cdot h_{ij} \qquad (6.7)$$

A useful measure that may be used to compare systems as to their cost in the reliability domain is X, which takes into account the cost for each failure mode and their rates of occurrence:

$$X = \sum_{j=0}^{M} (X_j \cdot R_j) \qquad (6.8)$$

6.2.4 Summary – Full Disclosure Report

The results of benchmarking experiments are most often used in the business sector to prove, that one's own product outperforms that of a competitor. Of course potential customers will find these results neither believable nor trustworthy without the possibility of replicating them. To make such replication possible, a *full disclosure report* describes in detail how the benchmark was conducted, what hardware and software was used, how the measures were taken and calculated. All information necessary to repeat the set-up and experimental phases (refer to Figure 6.3) must be included. Many performance benchmarks specifications already contain rules and guidelines governing the form and content of a full disclosure report. This section extends these guidelines for the information necessary to replicate the dependability part of the benchmark.

Table 6.0 summarises the measures and shows the additional items that must be included in a full disclosure report for a dependability benchmark. For each fault used in the faultload, the full disclosure report should contain a description on how this fault was applied. In addition to the performance measures, the full disclosure report should of course contain the dependability measures.

Table 6.0: Measures to be described in the full disclosure report

FMT: Failure Mode Table (obtained from benchmark experiments)

TDC: Table of Detection Costs (obtained from benchmark experiments)

A**:** The stationary availability (final measure)

X**:** The total failure cost of the system (final measure)

R_j**:** The occurrence rates of all failure modes used

Q_j**:** The repair rates of all failure modes used

S_A**:** The set of failure modes in which the system is available

S_U**:** The set of failure modes in which the system is unavailable

X_j**:** The total cost of each failure mode

The final measures are the stationary availability A and the total failure cost X. The other measures should be included because assumptions reflected in these measures are of interested to those reading the full disclosure report.

Finally, prose in any human language is often unclear and prone to misunderstandings. We strongly recommend that the set-up and experimental phases be described (perhaps additionally) using an unambiguous formal language. Annex 6-A elaborates this recommendation.

6.3 Specification of TPC-C-Depend

6.3.1 Introduction

The largest part of the TPC-C concerns the workload. The TPC-C workload is realistic and portrays the activity of a wholesale supplier. The workload includes the database design and layout as well as initial database population and a statistic framework describing the usage profile of the database. The workload does not change for TPC-C-Depend. An overview of the workload is given in 6.3.3. For dependability benchmarking, a faultload is needed in addition to the workload. As mentioned, a faultload may include different types of faults such as operator, software of hardware faults. The dependability benchmark described in this chapter focuses on hardware faults only.

In the rest of this chapter, all TPC-C refer to [TPC-C 2002].

The TPC-C devotes an entire clause to the description of the target system. This clause includes information about the system under test as well as the driver system and communications interface. For TPC-C-Depend we define in Section 6.3.2 how the system under benchmarking and the benchmark target relate to the system under test.

TPC-C provides information on the performance metrics calculated in clause 5. We extend this clause into the dependability realm (Section 6.3.5).

Clause 7 in TPC-C is devoted to pricing. The idea is, that to compare two systems fairly, the pure performance metric (such as transactions per minute) is not enough. If a system has a 10% better value for transactions per minute but costs three times as much as a competitor system, the amount of performance you get for your money is better for the competitor. For dependability benchmarking the pricing information must also cover maintenance and recovery actions.

In order for benchmark results to be considered compliant with the TPC-C [TPC-C 2002] benchmark specification, a full disclosure report is required. The intent of this disclosure is reproducibility, i.e. a team unrelated to the team that conducted the original benchmark should be able to replicate the results of this benchmark given the appropriate documentation and products. Clause 8 of the TPC-C includes a list of requirements for the Full Disclosure report. We extended this list in Section 6.2.4 and elaborate it in Section 6.3.4. We believe that a formal machine-readable description instead of a free-form report of the actions taken during the benchmarking experiment would greatly increase the possibility of reproducing the original benchmark results. Our approach is given in Annex 6-A.

Before the TPC accepts benchmarks as TPC-compliant and publishes them, an audit based on the benchmark's full disclosure report is performed. Rules concerning what an auditor must verify are included in the final clause of TPC-C. Some thoughts are given on how these rules should be extended, if TPC-C-Depend were to become part of the TPC audited benchmarking suite.

This section summarizes the documents which are part of the TPC-C-Depend and which are available as separate documents. Some familiarity with TPC-C is required for reading this section. It will focus primarily on the new aspects introduced when extending the TPC-C to create a dependability benchmark. Our purpose is to give an example of how an actual dependability benchmark can be derived from a well-known real-world performance benchmark.

6.3.2 System Under Benchmarking

For dependability benchmarking we distinguish between the *system under benchmarking* (SUB), and the *dependability benchmark target* [Dal Cin *et al.* 2002]. The latter is the part of the SUB, which is the focus of our interest. It is the system or component of the SUB that is intended to be characterized by the benchmark. In our benchmark, the BT is the software from Oracle and PostgreSQL.

In TPC-C-Depend SUB is identical to the SUT defined in clause 6 of [TPC-C 2002]. Figure 6.4 shows example 3 of the examples given in clause 6 of [TPC-C 2002].

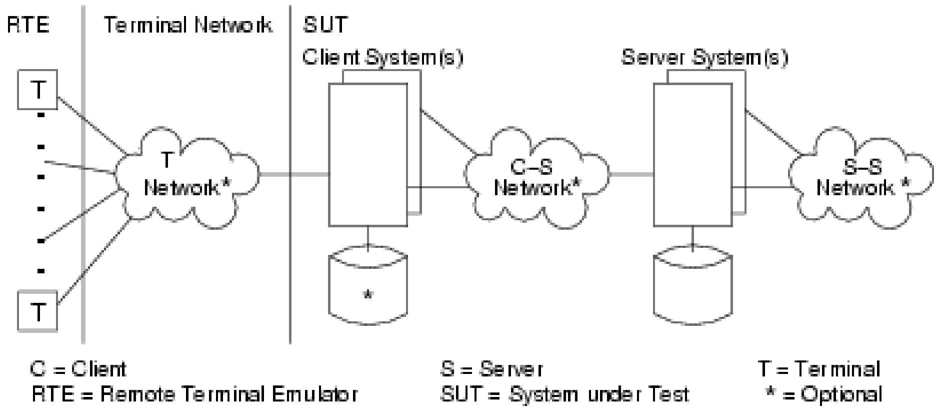

Figure 6.4: Model of a Target System (from [TPC-C 2002] clause 6.1)

The TPC-C defines the term *system under test* (SUT) in clause 6.3. According to the TPC-C, the SUT consists of one or more processing units, including both back-end and front-end systems, the communication network and data storage media. Hardware and software of these components are part of the SUT. The external *driver system* is defined in clause 6.4 of [TPC-C 2002]. The driver system is used to emulate the remote terminals including the users sitting in front of them. The driver system is necessary to generate the workload applied to the SUT.

The TPC-C takes dependability for granted. In clause 3 of [TPC-C 2002], four properties of transactions are defined: atomicity, consistency, isolation and durability. The SUT must meet these properties. The TPC-C includes tests for these properties. Since it is very important for a database system to comply with these four properties, we have used these to derive failure modes for the system under benchmarking.

The faultload may not directly modify the BT during the benchmark. No software vendor will accept a dependability benchmark of his software, where faults were introduced directly into his software during the benchmark. But everyone will agree, that software may run on faulty hardware or has to cooperate with other — possibly faulty — pieces of software (e.g. an operating system with third party device drivers) or that human error is possible (e.g. operator faults).

To evaluate dependability benchmarks, it is necessary to know the correct workload output of the system. For measures such as ”number of data items corrupted”, a comparison between the output of an experiment run and the correct output must be performed.

6.3.3 Workload

Performance and dependability measures can only be acquired, if the target system performs the services of interest to the user. In the TPC-C benchmark these are transactions related to the processing of orders in a fictitious wholesale supplier (clause 1.1). The TPC-C workload is a mixture of read-only and update-intensive transactions on the company's database that simulate the activities found in complex OLTP application environments. The following paragraphs give only an overview of the TPC-C workload. For detailed information, please refer to [TPC-C 2002] clauses 1 through 4.

The fictitious wholesale supplier has a number of geographically distributed and hierarchically organized sales districts and associated warehouses. Each regional warehouse covers 10 districts and each district serves 3000 customers. All warehouses maintain stocks for the 100000 different items sold by the company.

The company's database reflects this structure. There are tables to save information about warehouses, districts, customers and the history of business relations with each customer. More tables contain information about the items and current stock. Another three tables hold the data necessary to process orders. These nine database tables and their relationships are described in detail in [TPC-C 2002] clauses 1.2 and 1.3. The database must be populated before the start of the first experiment with randomly generated data ([TPC-C 2002] clause 4.3). After each single run the database is reset to this initial state. According to the scaling requirements detailed in [TPC-C 2002] clause 4.2, the size of the database tables in our benchmark are as given in the table in [TPC-C 2002] clause 4.2.2.

Fictitious employees of the wholesale supplier generate the transactions run on the database. In clause 2.2 defines what the user interface must look like, how data is entered and responses displayed. The fictitious users enter five different transaction types. These are defined in clauses 2.4 to 2.8. Four types of transactions are concerned with handling orders, entering a complete new order through a single transaction (new-order transaction), querying the status of an order (order-status transaction), registering a payment (payment transaction) or a delivery (delivery transaction). Finally, the stock-level of the warehouses must be controlled (stock-level transaction). The new-order, order-status and payment transactions must complete within a small timeframe to satisfy online users. The stock-level transaction has a relaxed timeframe requirement. The delivery transaction is executed in batch mode. The user only initiates the queuing of this transaction for deferred execution.

The original TPC-C also includes some erroneous user entries in its workload (new-order transaction, clauses 2.4.1.4, 2.4.1.5.1 and 2.4.2.3). The actions of the emulated users are described in clause 5.2 in every detail. The list of actions the user repeats circularly during the benchmark ([TPC-C 2002] clause 5.2.1) is as follows:

- Select one of the five possible transaction types from the on-screen menu according to a weighted distribution.
- Wait for the data entry form to be displayed.
- Measure menu response time.

- Fill in the input fields required for the chosen transaction type, taking some time to do this.
- Wait for the result to be displayed.
- Measure transaction response time.
- Think a while about the result.

The times the user takes to enter the data and think about the results are distributed as defined in clause 5.2.5.7. The response time is measured between two time stamps, the first of which is taken after the last character input by the user, the second after the last character output on screen. TPC-C Depend adopts this TPC-C workload and the measures.

6.3.4 Failure Modes

The TPC-C-Depend presented in this chapter focuses on a hardware faultload. Because of the fact, that the hardware used in the dependability benchmarking process is not known at the time the specification was written, we cannot define a hardware faultload in this section. The hardware faultload is therefore established during implementation of the benchmark. On the other hand, relevant failure modes are application domain specific only and can, therefore, be defined in the benchmark specification. TPC-C contains quite a lot of information concerning dependability-related properties of the system under benchmarking. The following list gives a short overview.

> **Atomicity:** Individual operations part of a single database transaction must either all be performed, or the system must assure, that no partially-completed transaction has any effect on data.
>
> **Consistency:** Assuming that the database is initially in a consistent state, no transaction may leave the database in an inconsistent state. The consistency conditions for the TPC-C workload database are listed in [TPC-C 2002] clause 3.3.
>
> **Isolation:** Modern databases can handle many transactions concurrently. Even if two concurrent transactions affect the same data items, data may not be left in an inconsistent state. Clause 3.4 of [TPC-C 2002] defines four isolation properties and four different isolation levels based on these properties.
>
> **Durability:** Clause 3.5 of [TPC-C 2002] defines this as the ability to preserve the effects of committed transactions and database consistency.

The durability property is clearly a dependability property. In clause 3.5.3, [TPC-C 2002] lists several single failures, which may not destroy the durability property. The list includes the following points:

- Permanent irrecoverable failure of any single durable storage medium containing TPC-C database tables or recovery log data.
- Interruption in processing that requires system reboot to recover.

- Failure of all or part of volatile memory.

If any benchmarking results for a system are to be published by the TPC, the system must fulfil the above requirements.

From the system properties listed above, we derive the following generic failure modes for the OLTP application domain:

Fully functional (FF): The following conditions must hold:

The injected fault had no observable effect according to the applied workload. The behaviour of the SUB is indistinguishable from its behaviour during the Golden Run, i.e. the SUB meets all the constraints concerning performance, transaction and system properties as specified in TPC-C [Sieh *et al.* 2002] clauses 3 and 5.

The SUB reports no errors of any kind.

All data remains consistent.

The SUB does not shut down.

Degraded Performance (DP): SUB behaviour does not fall into failure mode FF.

The SUB reports no errors of any kind.

All data remains consistent.

The SUB does not shut down or crash.

The injected fault causes a delay in response time, i.e. the SUB does not meet the 90th percentile response time constraint defined in TPC-C clause 5.2.5.7. The SUB must meet all other constraints concerning transaction and system properties as specified in TPC-C clauses 3 and 5. The 90th percentile response time constraints for this fault effect class are summarized in Table 6.1. This table is parameterised. The parameters α_x must be chosen during benchmark implementation and documented in the full disclosure report. The following constraints must be fulfilled:

$$\forall X \in \{NO, P, OS, D\} : 5 < \alpha_x \qquad (6.9)$$

$$30 < \alpha_{SL} \qquad (6.10)$$

NO, P, OS, D and SL refer to the New-Order, Payment, Order-Status, Delivery and Stock Level transactions respectively.

Table 6.1: Response Time Constraints for Fault Effect Class "Degraded Performance"

Transaction Type	**90th Percentile Response Time Constraint**
New-Order	α_{NO} sec.
Payment	α_{P} sec.
Order-Status	α_{OS} sec.
Delivery*	α_{D} sec.
Stock Level	α_{SL} sec.

* The response time is for the terminal response (acknowledging that the transaction has been queued), not for the execution of the transact-tion itself. At least 90% of the transactions must complete within 120 Seconds of their being queued (see [TPC-C 2002] clause 2.7.2.2).

Insufficient Performance (IP): SUB behaviour does not fall into failure DP.

The SUB reports no errors of any kind.

All data remains consistent.

The SUB does not shut down or crash.

The injected fault causes a delay in response time.

The system continues to respond to requests right up to the end of the measurement interval (i.e. the system does not hang).

Detected Error (DE): The SUB reports an error, either to the user, the operator or the error log.

All data remains consistent.

The SUB does not shut down or crash.

The SUB may or may not meet the response time or other constraints of TPC-C.

Shutdown on Error (SE): The SUB reports an error, either to the user, the operator or the error log.

All data remains consistent.

The SUB shuts down properly.

The SUB may or may not meet the response time or other constraints of TPC-C.

Shutdown (SD): The SUB does not report an error whatsoever.

All data remains consistent.

The SUB shuts down properly.

The SUB may or may not meet the response time or other constraints of TPC-C.

The term *system shutdown* is used to mean that the system shuts down properly, i.e. commits or rolls back transactions as required, closes open files, sends shutdown messages to connected clients, closes network sessions and after having done all this ceases to service any requests whatsoever until restarted.

System Crash/Hang (SC): The SUB does not report an error whatsoever.

Data may or may not remain consistent.

The SUB does not shut down properly but crashes/hangs. I.e. after a certain point during the measurement interval, the SUB no longer responds to any requests.

The SUB may or may not meet the response time or other constraints of TPC-C.

The term *system crash* is used to mean that the system ceases to service any requests whatsoever until restarted *without* the performing all the actions for proper system shutdown.

Bad Data (BD): The SUB does not report an error whatsoever.

Data is inconsistent in the database or inconsistent data is delivered to the user.

The SUB does not shut down.

The SUB may or may not meet the response time or other constraints of TPC-C [Sieh *et al.* 2002].

Unknown (U): The SUB behaviour does not fall into any of the above failure modes.

The SUB's behaviour during each experiment must be assigned to exactly one failure mode.

6.3.5 Measures and Measurements

The measures reported are the performance measures of the TPC-C for the Golden Run and the dependability measures already described in Sections 6.2.2.

The TPC-C reports two measures, the *maximum qualified throughput* (MQTh) and the 5-year pricing divided by the maximum qualified throughput. The MQTh is a number of orders processed per minute. The metric used to report the MQTh is the total number of completed New-Order transactions tpmC. The transaction response time (defined in Section 6.4.3) must be measured at the RTE (see Figure 6.4). The pricing measure is reported as price/tpmC.

In addition to the dependability measures, the tpmC during those failure modes of the systems that are in S_A is an interesting measure to report.

6.3.6 Benchmark Conduct, Procedures and Rules

The procedures and rules laid down in this benchmark specification are a mix of those defined in the TPC-C and those which are specific to dependability benchmarking and have been defined in Section 6.2.

For the full set of rules, procedures and requirements of TPC-C please refer to [TPC-C 2002]. The following examples from TPC-C should give you an idea, what rules are included in the TPC-C.

The TPC-C includes a number of rules to make sure, that a particular implementation is not specially tuned to give superior results in TPC-C while being quite useless in real-world surroundings. This includes tuning database settings especially for increasing TPC-C performance or using hardware and software that is not readily available to others. TPC-C considers pricing as suspect, when large discounts are given to a small set of potential customers, pricing features are a one-time special or are not documented properly. TPC-C will accept measurements that were gained using an approach, which is an accepted engineering practice or a standard.

As has been reported in Section 6.3.4 and detailed in Section 6.3.4, the TPC-C has a number of requirements concerning the behaviour of the system under benchmarking.

TPC-C devotes a complete section to detailing what must be included in the full disclosure report and what form this report should have. The additional information needed in the full disclosure report in case of a dependability benchmark is detailed in Section 6.2.4. Our recommendation is to use a formal language to describe at least the actions necessary during the benchmark set-up and experimentation phases. One approach to accomplish this is presented in Annex 6-A.

6.3.7 Formal Full Disclosure Reports to Ensure Reproducibility

Benchmarks are used to compare systems, often as marketing instruments. To ensure credibility, the exact benchmark configuration must be laid open in such detail, that others can reproduce the results and thus validate the benchmark. Ideally, all aspects of the benchmark configuration must be described semantically unambiguously.

Current professional benchmarks go through a lot of trouble to attain this mark. For the benchmarks of the Transaction Processing Council (TPC), the benchmark configurations are published as so called full-disclosure reports (usually several hundred pages in length), including detailed hardware descriptions as well as source code of the scripts and programs used. For benchmarks of the Standard Performance Evaluation Corporation (SPEC) such as the SPEC WEB99 [Linux kernel archives 2003] benchmark the same holds true.

Most benchmarking and fault injection experiments conducted without the goal to publish the results with organizations like TPC or SPEC who require strict compliance with their full disclosure policy never produce a detailed full disclosure report which would enable an independent team to validate the results. Everyone agrees on the fact that results which cannot be reproduced or validated in any way because the information on how these results were obtained is incomplete and can seldom be trusted.

Full disclosure reports have two tasks. On the one hand they describe the experiment set-up, environment, running and measuring in such detail, that an independent team can repeat the same experiment. On the other hand they include detailed information on the final results calculated from the measures.

To make it easier to create full disclosure reports which indeed hold all the information necessary to repeat the experiment described therein and reproduce the results, we suggest to use a semantically unambiguous formal language. This applies to the first task of a full disclosure report only. The final measurements should still be reported in the form of tables or graphs or whatever is appropriate for this type of measurement.

We recommend VHDL [IEEE 2002] as formal description language. VHDL has been in practical use in research and industry for a long time and its semantics for both static and dynamic descriptions are very well specified. Other modelling languages, such as the Unified Modelling Language (UML), which is being used by more and more people, are not suited for this purpose, as they do not yet have a well defined unambiguous semantic [2U Consortium 2003], [PUML 2003]. We use VHDL *only* as unambiguous description and modelling language. It is the essence of any description, that it places no demands on its implementation other that full compliance with the description.

Therefore, this approach implies at no point that the actual benchmarking experiment must be carried out using VHDL simulation! The experiment could be carried out using real hardware, a virtual machine or VHDL simulation. Of course, assuming that the description is indeed semantically unambiguous and that the implementations are carried out correctly, whatever the method employed, all must give similar results (within statistical limits).

An example of a disclosure report describing the experimental set-up as well as the experimental phase is given in Annex 6-A of this chapter. The static set-up description details the complete physical system, which may consist of more than one machine and can include interconnection networks. This static set-up is described in structural VHDL. Hardware alone is usually not enough to make a system useful. As a first step towards using the hardware, some sort of operating system must be installed. This is the case for desktop machines, servers and embedded controllers. For the latter, the operating system and the application providing the required services may be fused together. The dynamic set-up takes place during the experimental phase and is also described in VHDL.

Once the system is in production use and provides the required services, it responds to requests from its environment. When conducting a benchmark, the environment is different from a real-world environment and the stream of requests is usually generated artificially in some way. In a benchmark, the workload is the request stream arriving at the system interface and exercising the services the system provides. The workload configuration is described in behavioural VHDL.

6.4 Implementation Example

We have focussed on a hardware faultload. Any faultload, is, by definition, intrusive, since it changes the behaviour of the system. If a hardware faultload were to be implemented on real hardware, it would either be necessary to change (or add to) the hardware of the machine to inject (or simulate) hardware failures or to run additional software on the machine when SWIFI techniques are used.

We avoided both of these drawbacks by using a virtualisation environment (FAUmachine). The automatic experiment controller Expect simply turns the fault in a virtual hardware component on.

To detect the failure modes we observe the system from the perspective of the fictitious users, i.e. simply by noting the output of the system via it's defined interfaces but without interfering with the system. We also do a post-mortem analysis of all logfiles and the output of the server consoles. Being post-mortem, this analysis does not intrude on the system.

This section describes example implementations of TPC-C-Depend. First, the experiment set-up is described in Section 6.4.1. Section 6.4.2 defines the faultload used in this example. As was already explained, since the hardware is not known at the time TPC-C-Depend was written, the hardware faultload cannot be specified in TPC-C-Depend itself.

Section 6.4.3 presents and comments the results measured in our experiments. Section 6.4.4 details, how we have approached automation in our set-up. Sections 6.4.5 and 6.4.6 sum up how long it took to implement the example and to run the benchmark.

Remark: We also used some operator faults as in Chapter 5 and in [Vieira and Madeira 2002b]. These are not in fact part of TPC-C-Depend, but we included them in our experiments to test, how well other types of faults could be handled by our approach.

6.4.1 Experiment Set-up

This section details the experiment set-up. This includes the hardware and software used in the experiment as well as the benchmarking environment.

We have completed two example implementations with two databases, Oracle [Oracle Corporation 2002] and PostgreSQL [Postgresql 2003]. The two implementations use identical hardware and differ only in the database software used and other software changes necessary to accommodate the different databases. Whenever implementations differ, it will be mentioned explicitly. All general statements apply to both implementations.

6.4.1.1 Hardware Set-up

The SUB is a client-server system based on the model system shown in Figure 6-A.1 (Annex 6-A). Figure 6.5 shows an overview of the system used in the example implementation. The SUB is shaded light grey. The driver system, which is part of the Benchmark Management System, generates the workload and is shaded a darker grey. The SUB consists of two database servers (`db` and `rdb`) and four application servers (`as1` to as4). db is the primary database server, rdb is a reference server. The dependability benchmark target is the software (operating system and database software) running on db.

The reference server, which remains fault free, is needed to compare the output from db to the correct output. Because the workload is highly parallel and many parts of it are generated randomly it is not possible to determine data corruption by simply comparing the state of the database at the end of the Golden Run with the state of the database at the end of an experiment with a faultload for equality. Database files may differ, for example, simply because two transactions were executed in a different order in two different experiments and therefore have different timestamps! We therefore use the reference server and compare the result of each transaction as it is performed with the result of the primary server (which may be faulty) to detect data corruption.

The database and application servers are connected with an Ethernet local area network (solid lines in Figure 6.5). All servers have a serial terminal connected, which is used to take administrative actions as is necessary. The primary server also has a second serial terminal that prints alert messages from the database software for later analysis.

The application servers each have three additional serial terminals connected directly via a serial line. We do not have the additional (optional) T Network shown in Figure 6.4. These terminals are the end-user terminals. Simulated users sitting at these serial terminals generate the TPC-C workload.

Table 6.2: Node Hardware Set-up Differences

	RAM	hda	hdb	hdc	hdd	Serial 1	Serial 2	Serial 3	Serial 4
db	512MB	1GB	4GB R/W	8GB	1GB	x	x	o	o
rdb	512MB	1GB	4GB R/W	8GB	1GB	x	o	o	o
as1-as4	32MB	1GB	4GB R/O	N/A	256MB	x	x	x	x

The application servers receive the input from the users at the terminals and transform the user requests into SQL statements, which they send on to the database server for processing. When the response from the database server arrives, the application servers format the output and display it on the user's terminal.

The servers shown in Figure 6.5 all have the same basic hardware set-up. A graphical representation of the hardware set-up is shown in Figure 6-A.1, the differences are listed in Table 6.2.

6.4.1.2 Software Set-up

All machines use the Linux [Linux kernel archives 2003] operating system. The Linux distribution installed is Debian GNU/Linux 3.0r1 running kernel version 2.4.18.

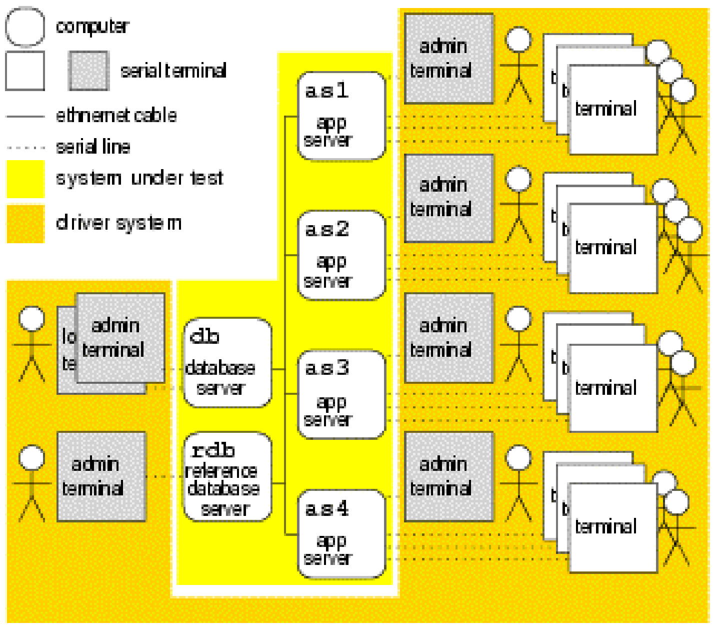

Figure 6.5: System Set-up Overview

We used Oracle 9i Database Release 2 Enterprise Edition [Oracle Corporation 2002] for Linux in one experiment. The changes made to the defaults of the standard Java Installer concerned the high-level network configuration (e.g. IP addresses) only. The database software is automatically started at system boot.

For the second experiment, we used PostgreSQL 7.4, which is distributed as a Debian package (named postgresql-7.4). As for Oracle, the standard set-up during installation is not changed except for high-level network configuration.

In both cases, the necessary data are distributed across several hard-disks (see Table 6.4).

hda: Operating system.

hdb: The database application, including software configuration and log files.

hdc: All the data files, redo logs and control files (database servers only).

hdd: Swap space only.

The reference database server is identical to the primary database server in all respects except for high-level networking, as the Internet Protocol requires that all machines reachable within a network have the same IP addresses.

The application server software is custom software written using the C programming language and Embedded SQL. Many databases have an API and libraries to supports Embedded SQL. SQL, the *Structured Query Language* [ISO/IED 1999] is an ANSI standard for data definition and data manipulation. The application server software is almost identical for Oracle and PostgreSQL. The only differences are the procedures for connecting with a database (which are not part of the SQL standard). For all other database operations, care was taken to use standard SQL and avoid special proprietary features of Oracle or PostgreSQL. This should also make it easier to port the application server software to other databases. The development of the application server software initially used the examples in Annex A of TPC-C. The necessary data are distributed across several hard-disks (see Table 6.2).

hda: Operating system.

hdb: The database application software, including the Embedded SQL libraries and any support libraries needed.

hdd: Swap space only.

During the set-up phase (see Figure 6.3) the database must be populated, i.e. loaded with randomly generated data that can later be manipulated by the virtual users in the TPC-C-Depend benchmark. The TPC-C specifies the table layout of the databases and how the data used to populate the databases must be generated.

The creation of the database, database user accounts and tables is quite database specific. To accomplish this task, SQL scripts are created, which are run by the database-specific SQL-interpreter.

First of all a database must be created to hold our data.

Next, we create a database user. If no database users are created, nobody could access the database. We create a single database user account called `tpcc` that is used by all virtual users. This account has enough rights on the database to read and write single entries.

For an Oracle database, before tables can be created, file space must be allocated to hold those tables. This space is commonly called tablespace. This step is not necessary for PostgreSQL.

Next the tables are created according to the table layouts given in TPC-C clause 1.3.

Finally we create indexes on the table containing the customer information and the one containing the order information. Indexes are usually created on database tables to speed up searches.

The database is populated using a custom program written in the C programming language and using Embedded SQL. This program is run a single time to populate the database according to the requirements of TPC-C. Apart from the code for connecting to the database, identical code is used for both databases.

Before an experiment is started, the state of the system is reset to the state at the end of the set-up phase.

6.4.1.3 Benchmarking Environment

The system described in the previous sections was set up using virtual x86 PC hardware provided by FAUmachine (formerly called UMLinux) [Höxer *et al.* 2002], [Höxer *et al.* 2002a], [Sieh *et al.* 2002], [Sieh *et al.* 2002a]. The original real-world software was installed on top of the virtual machines as described in Section 6.5.1.2. FAUmachine is Open Source software and freely available from FAU [FAUmachine Team 2003].

We decided to use a virtualisation environment to run our benchmark, because we intended to include hardware faults in our faultload.

Hardware fault injection techniques such as pin level fault injection and electromagnetic, laser and heavy ion radiation are often destructive or damage the system under test so that it cannot be reused for more tests. Hardware fault injection experiments are difficult to automate without an expensive set-up of supporting machinery. Literature about hardware fault injection includes [Gunneflo *et al.* 1989], [Arlat *et al.* 2003], [Madeira *et al.* 1994], [Sampson *et al.* 1998], [Steininger 1997].

Simulation based or software implemented fault injection (SWIFI) techniques have been used for quite a while to get around the drawbacks of using real hardware fault injection. The approaches mentioned here offer only the most cursory glance at this wide field. One approach is to inject faults into the data and code segments of a running process as well as into the registers and parts of main memory it uses. In addition, the system calls the process under test makes can be intercepted and their return values changed. FERRARI [Kanawati *et al.* 1992] is a tool which implements this and works fine for application processes. When using this approach to inject faults into the operating system itself, automating testing is difficult, since the computer must usually be rebooted manually when the operating system crashes or hangs. In addition, in the latter case the fault injection process should not be running on the machine with the (possibly already corrupt) operating system, since this can interfere with the integrity of the fault injection process and can thus lead to erroneous results. MAFALDA [Rodriguez et al. 1999] is a tool which implements a similar approach and injects faults into application service requests, data and code segments of micro-kernels of embedded systems. A fault injector using the `ptrace` interface has been presented in [Sieh 1993]. Other tools using SWIFI include FIAT [Barton et al. 1990] and Xception [Carreira et al. 1995].

The virtual x86 hardware FAUmachine provides is indistinguishable from real hardware from the point of view of any software (operating system or otherwise) running on this hardware. In addition, the FAUmachine environment has the great advantage of being fairly amenable to automation. When running the experiments, the virtual machines were distributed across two real machines:

1. 2.53 Ghz Pentium 4 processor, 1GB of main memory. The only virtual machine running here was the primary database server db.
2. 2 multithreading 3.06 Ghz Xeon processors, 2GB of main memory. All other virtual machines as well as the virtual terminals were running on this machine.

All real machines were running a Debian GNU/Linux 3.0r1 distribution.

The distribution of virtual on real machines was chosen, so that virtual machines don't become sluggish due to overloading of the real machine. We found, that the Xeon machine could easily handle the 5 virtual machines and the virtual serial terminals running on top of it.

6.4.2 Faultload

We have chosen to include only high-level storage hardware and network faults in the faultload for this experiment. Both storage and networking hardware are an integral part of a client-server database and OLTP system and are explicitly mentioned in the TPC-C document. As was mentioned previously, a complete faultload should include faults of all the different types. In this chapter we focus on hardware faults. We have also implemented some of the operator faults presented in [Vieira and Madeira 2002b] to validate, whether our approach is applicable to this type of faults, too. The software faults suggested in [Duraes and Madeira 2002a] can also be integrated in our approach.

For the final experiments we used the following set of hardware faults:

f_0 Permanent network send loss of the primary database server db. The percentage of packages lost varied between 1% and 100%.

f_1 Transient total failure of the data disk (hdc) of the primary database server db.

f_2 Power glitch, i.e. extremely short transient power failure (off/on) of the primary database server's power supply.

6.4.3 Results

This section summarizes the results we observed during our experiments.

Table 6.3 shows the failure mode Table for the PostgreSQL experiments, Table 6.4 the same table for the Oracle experiments.

Table 6.5 shows the fault attributes listed in Section 6.2.4. The rates r_i and q_i as well as the cost c_i are assumed values based on vendor information. The rates are given in failures/repairs per hour. The cost is given in monetary units (mu).

Table 6.6 shows the cost values assumed for detecting which fault caused the failure mode. For an explanation of the failure mode attributes in Table 6.7 refer to Section 6.2.3. Costs (C_j, X_j) are given in monetary units. The rates are per hour.

The availability set S_A includes only the two failure modes *FF* and *DP*. All other failure modes are members of S_U.

The total failure cost X of the system is 0.0018mu/h for Oracle, and 0.0021mu/h for PostgreSQL.

The Failure Modes:

Fully functional (FF)

Degraded Performance (DP)

Insufficient Performance (IP)

Detected error (DE)

Shutdown on Error (SE)

Shutdown (SD)

System Crash/Hang (SC)

Bad Data (BD)

Unknown (U)

The Parameters:

r_i**:** The fault rates of all faults used in the faultload (or MTBF instead).

n_i**:** For all faults used in the faultload, the number of times this fault was applied during an experiment.

n**:** The total number of experiments conducted.

q_i**:** The repair rates of all faults used in the faultload (or MTTR instead).

c_i**:** For all faults used in the faultload, the cost associated with repairing this fault.

C_j **:** For all failure modes, the cost associated with this failure mode.

Table 6.3: Failure Mode Table for PostgreSQL

Fault	**Failure Mode**								
	FF	**DP**	**IP**	**DE**	**SE**	**SD**	**SC**	**BD**	**U**
f_0 (send loss)	0.07	0.27	0.66						
f_1 (disk failure)	0.03			0.96				0.01	
f_2 (power failure)							1.0		

Table 6.4: Failure Mode Table for Oracle

Fault	**Failure Mode**								
	FF	**DP**	**IP**	**DE**	**SE**	**SD**	**SC**	**BD**	**U**
f_0 (send loss)	0.06	0.23	0.71						
f_1 (disk failure)	0.06			0.15	0.79				
f_2 (power failure)							0.98		0.02

Table 6.5: Fault Attributes (mu: monitary unit)

Fault	**Oracle**				**PostgreSQL**			
	n_i	r_i 1/h	q_i 1/h	c_i mu	n_i	r_i 1/h	q_i 1/h	c_i mu
Golden Run	20	---	---	0	20	---	---	0
f_0 (send loss)	372	10^{-5}	2	90	372	10^{-5}	2	90
f_1 (disk failure)	372	$0.5\ 10^{-5}$	1	300	372	$0.5\ 10^{-5}$	1	300
f_2 (power failure)	372	$0.6\ 10^{-6}$	7	10	372	$0.6\ 10^{-6}$	7	10

Table 6.6: Table of Detection Costs in mu

Failure Mode	Fault		
	f_0	f_1	f_2
FF	0	0	0
DP	30	0	0
IP	30	0	0
DE	10	10	0
SE	10	10	0
SD	50	50	0
SC	70	70	50
BD	0	30	0
U	70	70	50

Table 6.7: Table of Failure Mode Attributes

	FF	**DP**	**IP**	**DE**	**SE**	**SD**	**SC**	**BD**	**U**
C_j mu	0	10	30	20	70	70	200	200	200
R_j 1/h (PostgreSQL)	**8.5 10^{-7}**	**2.7 10^{-6}**	**6.6. 10^{-6}**	**4.8 10^{-6}**	**0**	**0**	**6 10^{-7}**	**5 10^{-8}**	**0**
Q_j 1/h (PostgreSQL)	0.17	0.54	1.32	0.96	0	0	7	0.1	0
X_j mu (PostgreSQL)	15.3	32.4	79.2	297.6	0	0	100	3.3	0
R_j 1/h (Oracle)	**9.10^{-7}**	**2.3 10^{-6}**	**7.1 10^{-6}**	**7.5 10^{-7}**	**4 10^{-6}**	**0**	**5.9 10^{-7}**	**0**	**1.2 10^{-8}**
Q_j 1/h (Oracle)	0.18	0.46	1.42	0.15	0.79	0	6.9	0	0.14
X_j mu (Oracle)	23.4	27.6	85.2	46.5	244.9	0	98	0	3

6.4.4 Automation

The virtualisation environment FAUmachine also includes Expect, an Automatic Experiment Controller. Expert could be considered as another part of the Benchmark Management System. Expect can watch the monitors of virtual machines or terminals and can generate input via virtual keyboards and mice. In addition to this, Expect can also trigger hardware faults in the virtual hardware. Currently most of the actions necessary during the set-up phase of the benchmark are done via Expect. The inputs to the set-up phase are currently the following:

- Debian GNU/Linux 3.0r1 installation CDs.
- PostgreSQL and Oracle installation CDs. These CDs were generated from the packages download from the Oracle or PostgreSQL sites respectively.

- CD containing the application server software for Oracle or PostgreSQL. This CD was generated from the custom application server software source code.
- the virtual hardware

Expect can handle the first time installation of the virtual machines automatically. Expect parses scripts that describe the virtual hardware and any actions like inserting a CD into the CD-drive, clicking or moving the mouse or entering text. Before Expect begins executing the actions it will first instantiate the virtual hardware according to the description. The formal language used for the Expect-scripts is VHDL, because it is well suited to describing static set-up and dynamic behaviour. VHDL also has a well defined semantic. In addition, the VHDL scripts that were written with automation in mind can be reused in a formal full disclosure report (described in Annex 6-A) as is, without any additional changes!

We also use Expect in the experimentation phase of the benchmark to automatically run the experiments. Expect also handles the complete generation of the workload. Instead of having a number of small programs run that model the fictitious users of the database and generate the workload, we include the workload description in the Expect scripts. The same holds true for the faultload. Since at the moment Expect only handles a single experiment at a time, there are a few shellscripts, which start Expect in an infinite loop to run a large number of experiments automatically.

Currently, the analysis phase of the benchmark is still handled using shell scripts to evaluate the data gathered by expect. This data includes the logfiles and console output of the servers and a trace of all actions initiated by the fictitious users. This trace includes the start and stop time of all transactions requested by each user, any non-standard actions (such as a reboot when the server does not respond for a long time).

6.4.5 Effort Needed

This section summarizes the effort needed to implement TPC-C-Depend. This basically includes preparing all the inputs necessary for the set-up phase (as shown in Figure 6.3).

Since TPC-C-Depend is an extension of TPC-C, you will need at least as much effort as is needed for implementing a full TPC-C benchmark. Depending on how much experience the benchmarking team has with TPC-C, and whether you are implementing the benchmark from scratch or adapting a previous implementation, this may take several weeks to only a few days.

To implement TPC-C-Depend, some additional effort is necessary during the set-up phase. You must collect data such as fault rates for the components in the hardware you are benchmarking and select the relevant faults based on this data . Once the faults to be included in the faultload have been selected and the workload implemented, a number of experiments are conducted during the experimentation phase.

If you intend to publish the benchmarking results in a TPC-compatible form, additional effort is needed to create the full disclosure report. To ensure that no crucial information is omitted, the detailed description of the actions taken during the set-up and experimentation phases should already be recorded, while you are actually performing these actions.

In Annex 6-A we show the benefits of using a formal language to record the actions taken during the set-up and experimentation phases in the full disclosure report. We show that VHDL is a language suitable for the task. The time needed for creating VHDL descriptions of the set-up and experimentation phases heavily depends on the level of experience the benchmarking team has with VHDL, and whether the VHDL descriptions are created from scratch or adapted from previous benchmarking activity.

In the benchmarking environment we used, creating the VHDL description has the additional benefit, that the same scripts can be used to automate the set-up and experimentation phase.

6.4.6 Benchmark Execution Duration

The benchmark execution duration includes the time needed to run through all three phases of the benchmark. Preparing the input for the set-up phase is not included in benchmark execution duration.

The set-up phase includes complete installation of operating system on all servers, installing database and application server software as applicable and populating the databases. With our set-up the time needed for the set-up phase was about 40 hours for Oracle and 7 hours for PostgreSQL.

Since the TPC-C requests, that the workload should run for at least 20 minutes, the minimum time for each experiment is 20 minutes. In addition there will be some overhead for resetting and booting and for recording the measures taken. In our experiments this overhead added up to about 10 minutes per experiment, for a total of 30 minutes time per experiment. The additional overhead is of course strongly dependent on the benchmarking environment, its speed and its degree of automation. The latter is especially important during the experimentation phase, as the same actions have to be performed a large number of times. Automation can therefore help reducing the manpower necessary to conduct the experiments. In our benchmarking environment, Expect controlled the experiments, so hardly any human intervention was needed during the experimentation phase.

Running the evaluation scripts will only take a few minutes. Some effort is needed to identify the failure mode caused by an injected fault. Again, the effort needed here is strongly dependent on how familiar the benchmarking team is with the database used (especially with error and log messages) and whether work from previous benchmarking activity can be reused.

6.5 Benchmark Validation

This section discusses how well TPC-C-Depend fulfils the properties listed above. We consider both the benchmark specification (the TPC-C-Depend document) and the implementation.

6.5.1 Representativeness

All benchmarks are necessarily an abstraction of a given application domain. TPC-C-Depend targets the application domain on-line transaction processing. TPC-C, the basis for TPC-C-

Depend, is an accepted industry standard and its workload has been considered as representative for the OLTP application domain.

We focussed on developing a representative hardware faultload. As has been discussed before, because the hardware used in the benchmarking experiment is not known at the time the benchmark specification is written, a hardware faultload cannot be included in the specification.

We have therefore decided to include in the specification a set of failure modes. The selection of failure modes is based solely on the application domain. To develop this set of failure modes we ran different hardware faultloads to test the full extent of possible system behaviour of both Oracle and PostgreSQL.

It may be possible, that our set of failure modes has to be extended when different types of faults, such as software and operator faults, are included in the faultload. However, in our experiments we did not observe behaviour falling outside the selected set of failure modes.

6.5.2 Portability

Portability concerns both the benchmark specification and implementation. Defining a general benchmark that is portable across application domains, e.g. one applicable to both OLTP systems and embedded space applications is not a viable option. Such a benchmark cannot take into account even the most general items typical for a certain application domain and would therefore completely miss the target of being representative for a certain application domain.

Of course a benchmark should be portable within its application domain. For an OLTP benchmark this means that an implementation can use any hardware, any operating system and any database software fulfilling the requirements of the application domain and implement the benchmark on this set-up.

TPC-C has proven for many years that it is portable across a wide range of databases, operating systems and hardware. The failure modes, which are part of the dependability extensions of this benchmark, are independent of the database, operating system or hardware used. These are therefore portable. The same holds true for the additional rules and procedures introduced in TPC-C-Depend.

It is a great advantage, if an implementation of a benchmark on one set-up is transferable to a different set-up without a lot of changes. We have originally implemented TPC-C-Depend with Oracle running on Linux. The transfer of the C-code for the application servers and the populator to PostgreSQL required very little change. The port of the code to MySQL (http://www.mysql.org) has also been finished and also required only little changes in the known problem locations. The same holds true for the workload, the only changes being necessary for the login procedures of the database. No changes whatsoever where required in the faultload implementation.

6.5.3 Repeatability and Reproducibility

Section 6.3 discussed the use of a formal language to describe the actions necessary during the set-up and experimentation phases. In our case, because we are using the benchmarking environment FAUmachine, the scripts used to automate the set-up and experimentation phases are the same ones reproduced in the full disclosure report. Different people of our team have used the automated scripts to run the benchmarks.

Because FAUmachine is a virtualisation environment, those measures dependent on time, depend on the speed of the real machine. Therefore, unless the real machines used have similar capabilities, those measures will differ. In the case of TPC-C-Depend this concerns the measures tpmC and all measures derived from tpmC. The dependability measures, such as the frequency of occurrence of certain failure modes are not dependent on time and therefore reproducible no matter the capabilities of the real machine.

Reproducibility concerns the benchmark specification. If two teams implement the specification using the same set-up (hardware, software, operating system), both implementations should give similar results, even if the teams do not communicate at all. We were unable to find a team with enough time on their hands to familiarize themselves with our benchmarking environment and implement TPC-C-Depend. Therefore we have not yet tested the reproducibility of our benchmark specification.

6.5.4 Scalability

OLTP systems come in all different shapes and sizes. TPC-C acknowledges this in clause 4, which contains the scaling rules and requirements for the workload. Since TPC-C-Depend does not change the workload from TPC-C, the scalability does not change. TPC-C has shown in hundreds of benchmark implementations on a wide range of different systems, that the scaling rules are valid.

For TPC-C-Depend we must consider the scalability of the faultload and the failure modes. The failure modes consider the behaviour of the complete system providing the OLTP performance. They are not concerned with individual parts and subsystems of the whole. Therefore, the failure modes scale automatically.

The hardware faultload is decided upon during implementation. A larger system will also have more hardware, so it is quite possible, that the faultload may include a greater variety of faults than the hardware faultload of a small system. A greater variety of fault types also means that more time is needed for experimentation. The experimentation time increases linearly with the number of different faults.

6.5.4 Non-Intrusiveness

Any faultload, is, by definition, intrusive, since it changes the behaviour of the SUB. If a hardware faultload were to be implemented on real hardware, it would either be necessary to change (or add to) the hardware of the machine to inject (or simulate) hardware failures or to run additional software on the machine when SWIFI techniques are used.

We avoided both of these drawbacks by using a virtualisation environment. The automatic experiment controller Expect simply turns the fault in a virtual hardware component on.

Annex 6-A : Full Disclosure Reports

A.1 Describing the Static Set-up

The static set-up description details the complete physical system, which may consist of more than one machine and can include interconnection networks. We recommend, that the finest granularity is the component level, i.e. we do not describe in detail the inner workings of commercial-off-the-shelf components, such as network interface cards, hard-disks or motherboards, but instead give enough information to identify such components unambiguously. Such identification information could e.g. be the manufacturer and the serial number.

The static set-up is described in structural VHDL

How a component fails is also part of the static set-up, whereas *when* a component fails is part of the dynamic set-up.

Figure 6-A.1 shows an example of a graphical representation of the structural VHDL-description of the hardware of a single machine. In a networking context, a single machine is often referred to as a *node*.

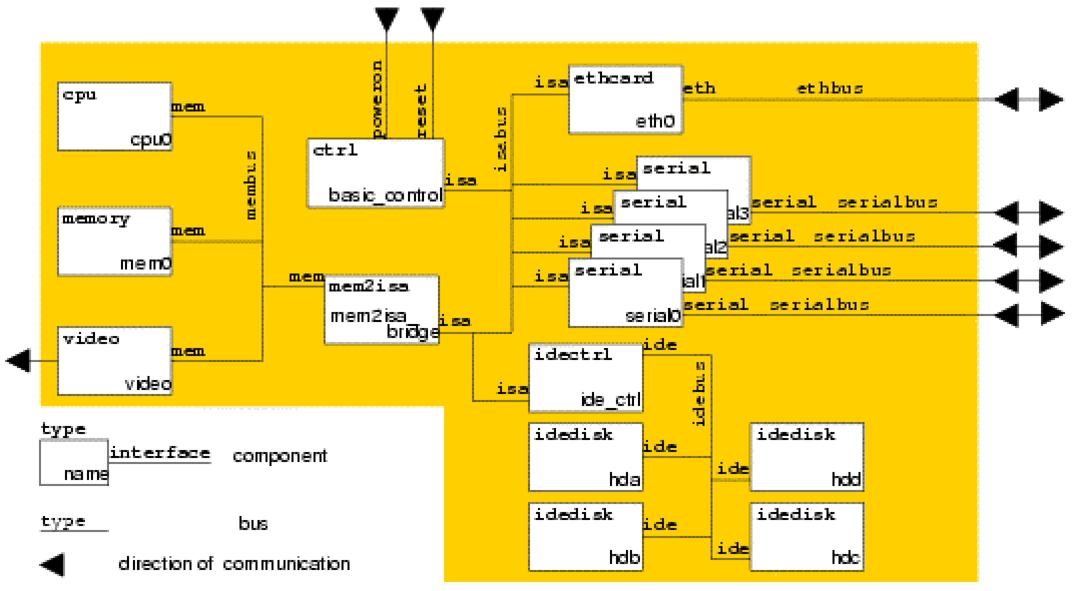

Figure 6-A.1: Graphical Representation of a Hardware Description in Structural VHDL

The node in Figure 6-A.1 is a typical server (such as kept in a 19 inch rack in a separate server room) without monitor and keyboard. CPU, memory, video card, the mem-to-isa bridge (commonly called northsouth-bridge), the IDE-controller, serial and Ethernet card are shown as separate components. They can be extra plugin cards or onboard components. As a server, the machine has four disks connected to the IDE controller. The `ctrl` component models the external switches (such as power and reset) on the machine's casing.

The links to the exterior world are through the Ethernet and serial interfaces and the buttons or switches (such as power-on and reset) present on the casing and the classical input/output devices keyboard, mouse and video. These interfaces to the outside world may be input only (such as the mouse interface usually is) or output only (such as non-touchscreen monitors) or input/output capable (such as the Ethernet or serial interfaces). Even if the interfaces are present, they are not necessarily connected (such as the interface of the video card of the server in Figure 6-A.1).

Once components such as the node in Figure 6-A.1 are defined, they can be used to build larger structures. To this end, the interfaces of the aggregate components are connected to appropriate buses.

A textual VHDL-description of a small client-server set-up with three nodes (two clients and a server) interconnected with a common network is shown in Figure 6-A.2. The Ethernet-interfaces of the nodes are connected to a common Ethernet (lines 2, 4 and 6 of Figure 6.6).

In this way we can build a hierarchy of components starting from the simple predefined ones and, reusing the components defined at the previous level of the hierarchy, finally arrive at very complex set-up descriptions completely in VHDL.

The component `node` (from Figure 6.3) is reused in Figure 6.4. It is therefore parameterised to allow flexible reuse. It is configured as server (large memory, line 4 in Figure 6-A.2) and client machine (smaller memory, line 6 in Figure 6-A.2).

For pure performance benchmarks the unambiguous definition of the configuration is all that is necessary concerning the hardware. For dependability benchmarks as proposed by DBench, we must take into consideration that hardware can fail and that these hardware failures may be part of a faultload. [Sieh *et al.* 1997] first proposed the idea of using VHDL-components with integrated fault descriptions. We are building on this general idea to define the failure behaviour of the basic components in the VHDL description.

Where the basic components used in [Sieh *et al.* 1997] are the classical low-level VHDL-components at the gate level, for dependability benchmarking we are using relatively high-level hardware components available off-the-shelf. The fault descriptions associated with such high-level components are of course not the same bit-flips and stuck-ats as those for gate level components. It is nevertheless possible to describe the component failure behaviour in VHDL. The following paragraphs discuss how to model possible faults using an Ethernet card as an example.

Figure 6-A.3 shows the VHDL description of the Ethernet card of the node in Figure 6-A.1 including its failure behaviour.

```
architecture structural of system is
   signal network : ethbus;
   begin
   server : node
      generic map (memsize => 512)
      port map (eth => network);
   clients : for i in 1 to 2 generate
      client : node
         generic map (memsize => 64)
         port map (eth => network);
   end generate;
end structural;
```

Figure 6-A.2: Structural VHDL Code for a Small Client-Server System

```
entity ethcore is
    port( isa : inout isabus,
          oeth : out ethbus,
          ieth : in ethbus );
end entity ethcore;

entity ethcard is
    port( isa : inout isabus,
            eth : inout ethbus );
end entity ethcard;

architecture internal of ethcard is
    fault ofault : boolean
       interval 1 year duration 5 sec;
    fault ifault : boolean
       interval 1 year duration 5 sec;
    signal oeth_int : ethbus;
    signal ieth_int : ethbus;
begin
    core : ethcore
        port map (isa => isa,
                  ieth => ieth_int,
                  oeth => oeth_int);

    ofault : process(oeth_int, ofault)
    begin
        if ofault then
            eth <= '0';
        else
            eth <= oeth_int;
        end if;
    end ofault;

    ifault : process(ieth_int, ifault)
        if ifault then
            ieth_int <= '0';
        else
            ieth_int <= eth;
        end if;
    end ifault;
end architecture internal;
```

Figure 6-A.3: Example of a VHDL Component Description Including Failure Behaviour

Figure 6-A.4 visualizes how the VHDL-components of Figure 6-A.3 are connected together. The VHDL entity `ethcore` is a black box, who functionality — behaviour compatible to that of the well-known NE2000 Ethernet interface card — is implemented in a component library. The `isa`-interface of `ethcore` is directly connected to the `isa`-interface of `ethcard`. The unidirectional interfaces `oeth` for output and `ieth` for input of `ethcore` are both connected to the bidirectional interface `eth` of `ethcard`. `oeth` and `ieth` are input signals to the processes ifault and ofault. Each process has another input signal, namely a fault signal. The fault signals are of the special VHDL type `fault`. A `fault` is defined in [Sieh *et al.* 1997] as a signal with a Boolean value that occurs with a certain frequency. The interval given is the mean time to failure. The duration is the mean fault activation time. These parameters are used for documentation only and should ideally be taken from known failure statistics of the component in question. They can later be used to calculate the measures of interest from the data collected in the experiment. Processes defined within the same `architecture` block in VHDL (as is the case with the processes ifault and ofault in Figure 6-A.3) are concurrent.

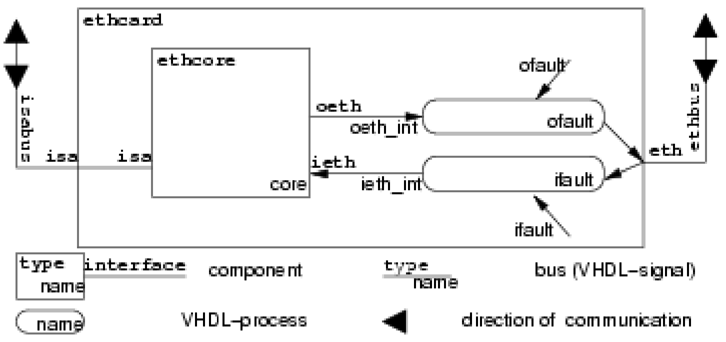

Figure 6-A.4: Graphical Representation of the VHDL Code in Figure 6.5

Actions within a single process are sequential. Whenever data arrives on the `eth` interface of `ethcard` the process ifault becomes active. Depending on the current value of the ifault fault-signal, the data is or is not forwarded on the `ieth` signal. Unlike the signals which represent physical buses or network connections, the fault signals are not connected to anything in the VHDL hardware description. When implementing the VHDL-description, they must be accessible to the entity responsible for running the hardware faultload.

The single important aspect of this approach is the semantic unambiguity of the VHDL-description, which is due to the fact, that VHDL is a syntactically and semantically very well defined standard. The VHDL description can then be implemented using real hardware or a simulation environment.

The *implementation* of the failure behaviour is a different matter altogether. Any fault injection method can be used to reproduce the component's faulty behaviour as described in the VDHL-code.

A.2 Describing the Dynamic Set-up

Every system exists within an environment and interacts with this environment. Depending on the system, the environment will be very different. If the system is an embedded controller, the environment with which the system interacts may consist of sensors and actuators. If the system is a web server, it will interact with administrators (usually human) and client machines. The requests issued from the client machines are often initiated by humans interacting with a web-browser. Even though we will use the term *user* in the following discussion, everything said also applies to other environments.

Section 6-A2.1 discusses the actions necessary during the set-up phase (refer to Figure 6.1) to prepare the system with the software needed to deliver the required services.

Section 6-A2.2 describes the dynamic set-up needed during the experimental phase (refer to Figure 6.3). The workload that the system must be prepared to handle during the benchmark as well as the faultload must be specified. The observer is necessary during the experimental phase to classify the systems behaviour according to the failure modes and is also part of the dynamic set-up.

A.2.1 Describing the Set-up Phase

Hardware alone is usually not enough to make a system useful. As a first step towards using the hardware, some sort of operating system must be installed. This is the case for desktop machines, servers and embedded controllers. For the latter, the operating system and the application providing the required services may be fused together.

The software configuration of a node is defined unambiguously by naming the input software necessary (e.g. ”SuSE 8.1 Professional”, ”Windows XP Home Edition”) and the actions necessary to install this software onto the system. The following list includes some of the actions that may be necessary during this part of software installation:

- understand output of the system (most often screen output)
- provide input to the system, e.g. by typing on the keyboard or pointing the mouse to a certain area on screen and clicking a mouse button
- insert and eject storage media such as DVD, CD-ROM, floppy
- pressing the power or reset button of the system (e.g. to reboot the system if necessary)

The bulk of the set-up information, such as the exact version (including patches) of the software installed is contained in the information, which installation medium is chosen. This part is static. The dynamic part, which is under direct control of the user, is the installation procedure. The necessary install actions can be described in behavioural VHDL. Figure 6.9 shows a part of such an installation description as a simple example. For most software there are less than 50 actions to take during installation if the defaults are accepted most of the time.

Two fragments of a graphical installation procedure are shown in Figure 6-A.5. In the first, the keyboard layout must be selected from a list, then the OK button must be clicked (lines 5 to

10). Later during installation, a root password must be entered twice (lines 11 to 16). The library functions `waitX button` and `waitX text` used in the example in Figure 6.7 take the expected output as a parameter and observe the actual output until either a timeout or the expected output occurs.

```
1  graphic_setup : process
2      variable result : boolean;
3  begin
4      setup_done <= false;
       ...
5      waitX_text(display, x, y, "Select Keyboard Layout");
6      move_mouse(mouse, x, y);
7      click_mouse(mouse, 1);
8      waitX_button(display, x, y, "OK");
9      move_mouse(mouse, x, y);
10     click_mouse(mouse, 1);
       ...
11     waitX_text(display, x, y, "Enter root password");
12     move_mouse(mouse, x, y);
13     click_mouse(mouse, 1);
14     send_keyboard(keyboard, "secret" & lf);
15     waitX_text(display, x, y, "Repeat root password");
16     send_keyboard(keyboard, "secret" & lf);
       ...
17     setup_done <= true;
18  end process graphic_setup;
```

Figure 6.A.5: Part of a Set-up Specified in Behavioural VHDL

In the case of `waitX text` the expected output is a text appearing on the given display at the given coordinates. `waitX button` waits for a button with a given label appearing on the given display at the given coordinates. `move mouse` moves the mouse to the given coordinates (simulating a user moving the mouse), `click mouse` simulates the user pressing and releasing the given mousebutton. `send keyboard` simulates the user typing the given characters on the keyboard.

After the operating system is functional, the application software necessary to provide the required services must be installed and/or configured. The actions the user must take are similar to those explained above. Software installation and configuration must be finished before the system is used in the experimental phase.

A.2.2 Describing the Experimental Phase

Once the system is in production use and provides the required services, it responds to requests from its environment. When conducting a benchmark, the environment is different from a real-world environment and the stream of requests is usually generated artificially in some way.

In a benchmark, the workload is the request stream arriving at the system interface and exercising the services the system provides.

The workload configuration is also described in behavioural VHDL. Figure 6-A.6 shows the small fraction of a workload for a database benchmark, which logs into the application server with the username `Alfred` (lines 7 and 8 in Figure 6-A.6) and starts the application server software (lines 9 and 10 in Figure 6-A.6). An attempt to login and start the application server software is only made, if the software has not already been started and both the database server and the application servers are already booted (line 5).

```
1 start_as : process (booted_as, booted_db)
2   variable started : boolean := false;
3   variable result : boolean;
4 begin
5   if (not started) and (booted_db = true) and (booted_as =
true)     then
6      started := true;
7       send_string(serial_recv_as0, "alfred" & lf);
8       wait_string(result, serial_send_as0, "Last login:", 1
min);
9       send_string(serial_recv_as0, "export TWO_TASK=db1.linux"
& lf);
10      send_string(serial_recv_as0, "/opt/bin/db/server" & lf);
11      started_as <= true;
12   end if;
13 end process start_as;
```

Figure 6-A.6: Part of a Workload Specified in Behavioural VHDL

`send string` and `wait string` are two library functions, which simulate a user typing something on the keyboard and reading something from the console respectively. `wait string` can be given a timeout. If the string waited for is not found before the timeout occurs, `result` is set to `false`.

After the system has been set-up, the workload can be applied. In this paper, we assume, that the system has interfaces, be they keyboard, mouse and monitor or network connection, which enable it to communicate with the outside world. The workload can in theory be applied by a human (carrying a precise watch), reading through the behavioural VHDL and executing the VHDL-instructions at the right times. More likely, to automate the benchmarking process, the VHDL is handcrafted into program code, which can be compiled and run. This is very easy for workload applied through a network interface, such as requests for a webserver. Another possibility is to have a program, which can read and interpret the VHDL instructions and execute them (much like the human mentioned above). Since mere software cannot operate physical devices (e.g. press keys on a keyboard) a virtualization environment must then be provided.

A faultload can contain hardware, software or operator faults. All of these can be described or applied in behavioural VHDL.

To inject a hardware fault during an experiment, we simply switch the affected component from its intact mode to its faulty mode in the VHDL code. To switch the example Ethernet card into its faulty mode, either one of the signals `ofault` or `ifault` would be set to true.

For a dependability benchmark it may be of interest to introduce design flaws into software in the environment of the application of interest. Depending on the application of interest, such software could be drivers [Duraes and Madeira 2002a] or the operating system. Flawed software can be installed or the flaws introduced into the software during the software set-up process using behavioural VHDL.

Apart from the hardware and software faultload described above, a faultload can also be applied to the system through its interface with the environment. All interaction faults can be applied in this way. Interaction faults are very diverse and may include operator faults (if the system of interest is a complex server), user input (for application programs), non-existent assembler instructions (for a CPU benchmark) or faulty input from sensors (if the system of interest is an embedded controller). Some examples are given in [Carrette 1996], [Kropp *et al.* 1998], [Miller *et al.* 1995], [Vieira and Madeira 2002b].

It is, moreover, possible to describe the expected output in behavioural VHDL and also to compare it to the actual output observed (or even not observed) in the current experiment. The functions `waitX button`, `waitX text` and `wait string` from the examples in Figures 6-A.5 and 6-A.6 take the expected output as a parameter and observe the actual output until either a timeout or the expected output occurs. The expected output in this case is the required output, but it can also be a known error message, for example. Since the behavioural VHDL already describes how to observe the output and control the workload, it can also describe when to collect the data necessary for calculating the measures. For a dependability benchmark based on SPEC WEB99 [Linux kernel archives 2003], the VHDL description could contain statements to count the number of requests initiated, the number of successful requests and the number of aborted requests, for example.