Software Functional Fault Injector for SDDS

Grzegorz Łukawski, Krzysztof Sapiecha

Department of Computer Science
Kielce University of Technology
Al. 1000-lecia PP 7
25-314 Kielce, Poland
g.lukawski@tu.kielce.pl
k.sapiecha@tu.kielce.pl

Abstract: Scalable Distributed Data Structures (SDDS) consists of two components dynamically spread across a multicomputer: records belonging to a file and a mechanism controlling record placement in file space. Record placement mechanism (RPM) is spread between SDDS servers, their clients and dedicated process called split coordinator, running on any SDDS server. To investigate fault tolerance of RPM a software implemented fault injector (SWIFI) specialized for SDDS was developed. In spite of regular SWIFIs, where bit flip fault model is used, in SDDSim a functional fault model was assumed. This results in high performance of the simulator not diminishing its informative value. Then, an experiment consisting of injecting faults into RPM was executed. The experiment proved high vulnerability of SDDS on the faults. Finally, the SDDSim was supplemented with options that made it possible to extensively analyze different features of SDDS with special emphasis placed on its fault tolerance.

1 Introduction

As computer systems become more and more complex system designers have to deal with many different failures and various error detection and fault tolerance mechanisms incorporated both into hardware and software of the system. A key task for the designers is to analyze fault effects for the system. This can be done experimentally with the help of so-called fault injectors (FIs). FI injects faults into a system and then monitors the behavior of the system.

Physical fault injectors are accomplished by corrupting logic values at circuit pins, disturbing voltage on power lines, bombarding the hardware with heavy ion radiation etc. These techniques need special and expensive equipment. Moreover, a class of faults which could be injected into a system and a range of an experiment are limited.

Software implemented fault injectors (SWIFI) disturb states of CPU registers, data and code memory locations. SWIFI do not need any special equipment and are more flexible than hardware implemented ones [CMS98].

Scalable Distributed Data Structures (SDDS) [LNS96] is a multicomputer application but very specific one. It consist of two components dynamically spread across a multicom-
puter: records belonging to a file and a mechanism controlling record placement in file space. Record placement mechanism (RPM) is spread between SDDS servers, their clients and dedicated process called split coordinator (SC), running on any SDDS server. Two factors are thus crucial for SDDS dependability: fault-tolerance of data stored in records and fault-tolerant record placement in file space. When we studied these aspects of SDDS we discovered that there are many techniques making SDDS tolerant to faults of data [LN96], but there is none that makes SDDS tolerant to faults of data placement. The question aroused then how dangerous the latter faults are for SDDS. To answer the question a SWIFI specialized for SDDS (SDDSim) was developed and an experiment consisting of injecting faults into RPM of SDDS was executed [SL05a]. The experiment proved high vulnerability of SDDS on the faults considered. Hence, dependable RPM was worked out [SL05b, SL05c]. Finally, the SDDSim was supplemented with options that made it possible to extensively analyze different features of SDDS with special emphasis placed on its fault tolerance.

In the paper SDDSim architecture is presented and most interesting features of regular and fault tolerant SDDS schemes are given. In section 2 a brief description of SDDS is given. The functional fault model is introduced in section 3. Our software functional fault injector is presented in section 4. Experimental results are given in section 5. The paper ends with conclusions.

2 Scalable Distributed Data Structures

A record is the least SDDS element. Every record is identified with a key unique for a file. Records equipped with keys are loaded into buckets. The buckets are stored on machines (computers - multicomputer nodes) called the servers. Every server stores one or more buckets. All the servers can communicate with each other. If a bucket’s capacity is exceeded it performs a split, moving about half of the records to a new bucket.

Any number of machines called clients can simultaneously operate SDDS file. Each client does not know anything about other clients’ existence. Every client has its own file image (information about an arrangement and the number of buckets), not exactly reflecting actual file state. A client may commit addressing error, sending a query to inappropriate bucket. If that happens, the server forwards such query to appropriate bucket and the client receives Image Adjustment Message (IAM), updating this client’s file image near to actual state. A client never is going to commit the same addressing error twice.

There is a single dedicated process called split coordinator (SC), controlling buckets’ splits, communicating with servers only. The clients, servers and the split coordinator are connected together with a net.

The SDDS file grows as the amount of required storage space increases1. There is no need for central directory. Splitting and addressing rules are based on a modified linear hashing method (LH*) [LNS96]. The rules may be implemented in two ways: centralized (the SC

---

1If all the buckets are stored in RAM memory then SDDS highly improves data access efficiency [LNS96].
is used) and decentralized (no SC, a *split token* is used instead). More detailed description of LH* and its variants may be found in [LNS96].

Two factors are crucial for SDDS dependability: fault-tolerance of data stored in records and fault-tolerant record placement in file space. There are many techniques making SDDS tolerant to data faults, using different kinds of redundancy, from full file mirror to Reed-Solomon codes [LN96]. Methods of making SDDS tolerant to control faults are given in [SL05b, SL05c].

## 3 RPM functional fault model

The following functional faults may occur in RPM [SL05b, SL05c]:

*The client:*  
- "Client turned Berserk” before performing any operation. Such a client sends each query into randomly chosen bucket, bypassing the file image.
- "File image cleared” (set to represent empty file) at any moment. First query is sent to the bucket 0 (according to the SDDS LH* principles [LNS96]), the client updates the file image then.

*The bucket:*  
- "Add/remove split token(s)” at any moment (for decentralized SDDS scheme). So there may be none or multiple tokens in the SDDS file at the same time.

*The SC:*  
- "SC turned off” (switched to 'Deaf’ mode) - every incoming message is ignored then. May be turned on again too.
- "SC turned Berserk” at any moment. Such SC sends split messages to randomly chosen buckets.

## 4 Software fault injector

To study SDDS behavior under fault conditions a specialized SWIFI, called *SDDSim*, was developed. It is an application using SDDS file for storing integer key values. The file is able to expand only. Main differences between SDDSim and a multicomputer SDDS are the following:

- The buckets are stored in local PC’s RAM instead of multicomputer global RAM.
- There is no real data stored in each of the buckets.
Table 1: SDDSim operating modes

<table>
<thead>
<tr>
<th>SC mode</th>
<th>Description</th>
<th>Fault tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No SC - token mode.</td>
<td>None</td>
</tr>
<tr>
<td>0+</td>
<td>No SC - token mode with token tracing.</td>
<td>Token tracing</td>
</tr>
<tr>
<td>1</td>
<td>Single SC - basic SDDS LH* mode.</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Double SC. Third SC run in case of any single SC failure.</td>
<td>JCT (TMR)</td>
</tr>
<tr>
<td>3</td>
<td>Three SCs run just as the file is created.</td>
<td>TMR</td>
</tr>
</tbody>
</table>

The application allows monitoring all SDDS file activities interactively. All operations are logged into a log file and can be analyzed later. Another application called SDDSlog was developed for this purpose. Such a solution leads to the following limitations:

- Only SDDS functionality is tested, time related problems are neglected.
- Each element (process) performance is almost equal to every other element one. On the contrary a multicomputer may be built from different machines having different amount of RAM memory, different network interfaces, etc.

4.1 Principles of operation

There are 5 modes of operation of SDDSim (Table 1). The first two are decentralized, while the last three are centralized control schemes. The user may choose the bucket capacity and the number of active clients, too.

SDDS file is placed in single PC. Every SDDS component runs as a separate process. According to SDDS scheme there are three types of processes:

- **The client** that represents real SDDS front-end. The client sends queries to the buckets using his own file image. There may be up to 3 clients. The user is free to choose any client at any moment. With the help of the clients, the user is able to perform three types of operations (insert, find and delete) with SDDS file.

- **The bucket** able to store 16 - 100 K key values. There may be at most 128 buckets. The bucket sends 'collision' messages as soon as it reaches critical load level (chosen by the user). The bucket is able to store 50% more records over the storage capacity. It stops accepting new data if the load reaches 150%.

- **The split coordinator** for centralized SDDS scheme. There may be at most 3 SCs (depending on SDDS scheme).

Each process has its own message queue implemented as a list. It is a priority queue with two priority levels:

---

2Upper limit of buckets could be altered.
1. *low priority* - most of the messages, including data transfers, collisions etc.

2. *high priority* - used for control messages processed before any low priority message.

The interprocess communication mechanism consists of single semaphore for each message queue, used for message counting and a wxMutex [wxW] object for mutually exclusive list access. Each incoming message is placed at the end of low or high priority message list respectively. If there is a high priority message waiting then it is processed first. Low priority messages are processed only when the high priority queue is empty. Every message consists of message ID, reply address and optional key value.

Faulty situations may be arranged at any moment of SDDS file evolution. Faults are injected at functional level, precisely into any SDDS component. Fault injection is performed by the user whenever he wants to. The control panel of each SDDS file component has a set of flags for fault injection. If the 'deaf' flag is set in SC type component, the component ignores every incoming message (goes deaf). If 'berserk' flag is set in client type component, the component sends messages to randomly chosen buckets and so on. The token is just another boolean variable. It may be set or clear for any bucket. Fault injection concerning decentralized SDDS scheme is done this way.

The bucket from another bucket’s point of view behaves as a client, so no fault injection concerning bucket’s address calculation is implemented.

### 4.2 Implementation

The fault injector is built using ANSI C++ and wxWidgets [wxW] framework. Such a solution brings many advantages:

- SDDSim may be compiled with almost any ANSI C++ compiler for different operating systems.
- Thoroughly tested code fragments may be used to build real SDDS implementation.
- The wxWidgets framework is not a commercial product, evolving as an Open Source for over than 12 years. It is implemented for many operating systems: Microsoft Windows, Linux (X-Window), Sun Solaris, Mac OS, OS/2.
- wxWidgets offers full set of classes and functions for convenient user interface building, independent from the operating system used.
- Unified set of classes and functions for running, managing and synchronizing processes, without the need of direct OS system calls.
- Unified file, graphics, standard input/output handling, etc.

Every component is divided into functional parts and put into classes. Common code was put into two base classes:
• **SDDSmessage** is used for message transporting as it was explained above.

• **DrawBox** for drawing component status into the main SDDSim window.

Each process class is derived from these two and *wxThread* class coming from the *wxWidgets* package. All the buckets (*SDDSBucket*), clients (*SDDSClient*) and SCs (*SDDScoordinator*) are managed with single *SDDSFile* object. Fault-tolerant components are derived from corresponding basic classes:

• **SDDSBucketCF** (Centralized and Fault-tolerant) and **SDDSBucketDF** (Decentralized and Fault-tolerant) from *SDDSBucket*;

• **SDDSCoordinatorCF** (Centralized and Fault-tolerant) from basic *SDDSCoordinator*. There is no SC in decentralized control mode.

5 Experimental results

All SDDSim operations are logged into files. These files are analyzed later using *SDDSlog* application, which is capable of generating graphs presented below. The horizontal axis shows the progress (total number of messages transferred), because no real time measurement takes place, as it was explained above. Vertical axis should be interpreted differently for three curve types:

• **Messages** - the vertical axis shows how much of the network throughput is taken by messages of given type (0-100%).

• **Number of buckets** - shows how many buckets are currently used (0-128).

• **Average file load** - computed for all active buckets (0-150%, according to the bucket behavior explained above).

Our analysis is focused on insert operations, because it is the only kind of client operation that increases file load, causes bucket splits and engages control mechanisms which we are going to test.

5.1 File evolution

A file having a single split coordinator was created, a series of insertions were performed then. The file has expanded reaching maximum number of buckets. File was inflated with series of succeeding key values (Fig.1), the average file load increases as more successful inserts are done and it decreases while some splits are performed. It keeps increasing if all the available buckets are used and there is no possibility to make another split.
Similar file was created again, inflated with the same number of key values, generated randomly this time (Fig.2). Some of the insert operations failed as a few of the buckets became overloaded and stopped accepting more records (Fig.3).

5.2 Insert versus transfer

During a split, around half of a bucket capacity is transferred into a new bucket. The network throughput is taken by transfer messages while split operation(s) takes place (Fig.4).
5.3 Fault-tolerant control schemes

A similar file was created. A first round of inserts were performed, the lonely SC was turn 'Berserk' then. The second round of inserts led to file structure corruption and many messages were lost (sent beyond file space), because SC was sending split messages to randomly chosen buckets, bypassing the LH* rules and file image (Fig.5).

A similar file was created again, supplied with two SCs this time. The same situation was arranged and fault-tolerant mechanisms proved their usability (Fig.6). There was no single lost message and the file continued to work properly.

6 Conclusions

SDDSIm is a single-computer application and as it was proved it behaves almost as real multicomputer-based SDDS. It is a useful tool for SDDS control mechanisms testing. In
spite of regular SWIFIs, where usually the bit flip fault model is used, in SDDSim a functional fault model was assumed. This results in high performance of the simulator not diminishing its informative value. SDDS schemes both regular and tolerant with regard to fault control were thoroughly tested. The experimental results were very informative. For example, theoretical estimations concerning decreasing SDDS efficiency due to control mechanism fault tolerance were verified experimentally with the help of SDDSim. 100 000 records were inserted into SDDS file working in two modes, first with single then with triple SC. The experiment was repeated 5 times and the best and the worst values were dropped. Finally, average time for each of the mode was calculated. It was 13.97s for the first and 14.02 for the other of the modes\textsuperscript{3}. This meant that the triple coordinator worked 99.64\% as efficient as the single one what agreed with estimated 99.7\% [SL05b].

As far as investigation of different aspects of fault tolerance is concerned, SDDSim has some advantages over real multicomputer SDDS. These are the following:

- Only one PC is enough to simulate complex SDDS file, consisting of hundreds of buckets and millions of records. File size is limited by single computer’s RAM memory size and CPU power.

- The simulation is free from faults caused by hardware components, network connection, operating systems, etc.

- The file may be set up to work in a few seconds without the need to engage many computers and personnel. The file may be shut down very fast too.

- Other network traffic doesn’t affect simulation results in any way.

- The simulated file works much faster than real SDDS, allowing the user to run many exhaustive tests in an acceptable time.

\textsuperscript{3}PC with AMD Athlon64 2800+ processor and 512MB of RAM was used.
Tests may be repeated over and over in exactly the same environment. All the actions are logged into a file and may be exactly analyzed later.

SDDSIm is a functional fault injector. Therefore, the time required for performing even complex experiments is rather short. On the other side the faults concern control mechanisms only, while generic SWIFI would also affect data stored in the buckets.

References


