PADI-DSTM

PADI
Middleware for Distributed Internet Applications

Abstract

The PADI project aims at implementing a simplified distributed software transactional system.

1 Introduction

The goal of this project is to design and implement **PADI-DSTM**, a distributed system to manage data objects that reside in memory and can be shared by transactional programs that execute in different machines. This will be a simplified version of a *Distributed Software Transactional Memory System*[5, 2, 3].

To simplify the work, and make the project doable in the timeframe available for the project, the applications will only share a particular type of object, of type *PadInt*, that stores an integer that can only be accessed by well defined *read* and *write* methods.

The work will start by writing a paper describing the solution . and only then should the system be developed sequentially in phases .

2 Architecture

The architecture of the **PADI-DSTM** includes a centralised master, a variable number of servers (that store data objects), and a variable number of clients that concurrently access one or more data objects transactionally. The clients run applications that are linked to the **PADI-DSTM** library and that access shared objects stored at the servers. Each server stores a set of shared objects. The servers' interfaces are identified by a URL in the format "$tcp : // < ip-address >:< port > /Server$". The students are free to implement the library as they wish; in particular, the implementation of the library methods may communicate with the centralised master if needed. Still, for scalability, the implementation should strive to use decentralised solutions as much as possible, such that the centralised master does not become a bottleneck.

The students may assume that the centralised master is always running and that it does not crash. Students may also assume that clients do not crash in the middle of the execution although they will terminate when the application reaches its end. Also, when testing and debugging the system they may assume that the master is launched before the servers. If a server does not find the master active it is not required to provide service.

3 The PADI-DSTM Library

The library to be created by the students is linked to each application that uses **PADI-DSTM**. The library exports a set of methods that provide access to **PADI-DSTM**:

- *bool Init()*: this method is called only once by the application and initializes the **PADI-DSTM** library.
- *bool TxBegin()*: this method starts a new transaction and returns a boolean value indicating whether the operation succeeded. This method may throw a *TxException*. A *TxException* should include a string indicating what caused the exception.
- *bool TxCommit()*: this method attempts to commit the current transaction and returns a boolean value indicating whether the operation succeded. This method may throw a *TxException*.
- *bool TxAbort()*: this method aborts the current transaction and returns a boolean value indicating whether the operation succeeded. This method may throw a *TxException*.
- *bool Status()*: this method makes all nodes in the system dump to their output their current state.
- *bool Fail(string URL)*: this method makes the server at the *URL* stop responding to external calls except for a *Recover* call (see below).
- *bool Freeze(string URL)*: this method makes the server at *URL* stop responding to external calls but it maintains all calls for later reply, as if the communication to that server were only delayed. A server in freeze mode responds immediately only to a *Recover* call (see below), which triggers the execution of the backlog of operations accumulated since the *Freeze* call.
- *bool Recover(string URL)*: this method makes the server at *URL* recover from a previous *Fail* or *Freeze* call.

The library also mediates the interaction with the shared distributed objects. Basically, it offers primitives to create a shared object and to obtain a reference to a shared object that was previously created. Each object is uniquely identified by an integer id. For simplicity, we assume that all shared objects are of type *PadInt*. The methods exported by the library regarding the local object manager are the following:

- *PadInt CreatePadInt (int uid)*: this method creates a new shared object with the given *uid*. Returns *null* if the object already exists.
- *PadInt AccessPadInt (int uid)*: this method returns a reference to a shared object with the given *uid*. Returns *null* if the object does not exist already.

Each shared object can only be accessed in the context of a valid transaction. For this purpose, an object of type *PadInt* exports two different methods:

- *int Read()*: reads the object in the context of the current transaction. Returns the value of the object. This method may throw a *TxException*.

- *void Write(int value)*: writes the object in the context of the current transaction. This method may throw a *TxException*.

A transaction may abort as a result of a read or write operation. In this case a *TxException* exception is triggered, whose internal contents (e.g. a string) should help in describing the cause of the transaction abort. Additionally, *TxException* exception should be triggered with informative string if somehow the input is considered invalid.

4 Consistency

The system should provide sequential consistency. The students should refer to the classes and to the bibliography of the course for the details of this consistency model[1, 4].

5 Fault Tolerance (optional)

The students must first implement a version of the system that is not required to tolerate faults. If, and only if, this version is operating correctly should the students consider implementing a fault-tolerant version of the system. The fault-tolerant version is only required to tolerate the failure of a single server node. Clients are assumed never to fail.

In the fault-tolerant version of the system each data object must be stored in at least two different server nodes. Also a server node should be able to crash without blocking or compromising the consistency of the data, i.e., if a server node crashes during the execution of a transaction the atomicity must be preserved: either the transaction commits before the crash and the updates will be visible to the other servers, or the server node fails before the transaction commits and none of the updates of the interrupted transaction become visible. Furthermore, whenever a server node commits a transaction, the updates must have been replicated (and ready to commit) at least in another server node, such that the results of the transaction survive a potential failure.

6 Papers

Students should begin the project by writing a paper (max. 4 pages) describing the architecture of the solution (software components, algorithms and protocols). In this paper, students should follow the typical approach of a technical paper, first describing the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. The paper should include an explanation of the algorithms used and justifications for the design decisions.

The described algorithms should include at least:

- distributed architecture overview;
- data structures for the main components of the architecture and their distribution;
- division of responsibilities among master and servers, with issues requiring the intervention of the centralised master;

- algorithms for concurrency control, e.g. locking;
- algorithms for transaction validation, commit;
- distributed protocols for the above and other aspects of the project;
- algorithms for deadlock detection and abort recovery.

The final report should be an extension of the paper that was previously submitted. The final report (max. 6 pages) should be as detailed as possible and include some qualitative and quantitative evaluation of the implementation. The quantitative evaluation should be based on reference traces that will be provided at the project's web site, and focus, at least, on the following metrics:

- metrics for evaluating algorithms and protocol performance: latency (number of round trips), bandwidth (total number of messages and size);
- overall throughput, influenced by the aspects above, and amount of parallel progress allowed in transaction execution;
- load balancing regarding data storage and processing among servers;
- violations in transactional enforcement: lost updates, inconsistent retrievals;
- amount of aborted transactions, proneness to cascading aborts;
- occurrence of or proneness to deadlocks.

Right from the first paper, this should also motivate a brief discussion on the overall quality of the algorithms and protocols proposed and developed.