BFT Services Programming with a Dependable Tuple Space

François Aubry, Alysson Neves Bessani, Paulo Sousa
University of Lisboa, Faculdade de Ciências, LaSIGE - Portugal

1. Introduction

In the last decade there has been a large body of work in how to make Byzantine fault-tolerant (BFT) systems more practical. Most of these works aim to provide efficient implementations of abstractions such as state machine replication, consensus and read-write registers. These abstractions are well regarded as important building blocks to architect dependable distributed systems.

In this paper we advocate the idea of using a new abstraction for programming dependable systems: the coordination service. The idea is to have one or more shared memory objects providing some synchronization power that allows distributed problems to be solved. The final objective is to make dependable services programming more simple. The idea is not only interesting from a theoretical point of view, but is also very practical since coordination services are already being used by Google and Yahoo! to coordinate crash fault-tolerant services.

In this paper we evolve this idea to deal with Byzantine faults and show how we can use a coordination service based on an abstraction called PEATS (Policy Enforced Augmented Tuple Space) [2] to implement BFT services. The PEATS is a simple object that provides operations to insert, read, remove and conditionally insert generic data structures called tuples. The execution of these operations is protected by a fine-grained security policy that allows or denies them based on the operation arguments, its caller and the state of the PEATS. Given the limited number of operations provided by the PEATS, it may seem hard to implement a system just by using it. However, these operations are powerful enough to solve any distributed computing problem [1], [2].

In a previous work [2], it was showed that the PEATS can be used to build BFT algorithms that are very efficient and almost as simple as crash fault-tolerant algorithms. In this paper we want to exploit this abstraction to build practical BFT services. To illustrate the effectiveness of our approach we present a set of algorithms to implement a LDAP-like Naming and Directory Service (NDS) using the PEATS abstraction. In order to do this, we start by defining a way to represent the directory tree structure using tuples. Then, we develop directory service’s algorithms using only the PEATS’ operations presented before.

2. A PEATS-based Naming Service

There are some fundamental challenges that must be addressed in order to implement a NDS over any coordination service. For a PEATS in particular, we have to solve two main problems: (1.) how to represent the hierarchical data structure of a NDS in the non-structured tuple space and (2.) how to implement the NDS standard operations efficiently. The solutions for these two problems are discussed in the next sections.

2.1. NDS Data Structure

A namespace of an LDAP-like NDS is a tree structure in which nodes are directories or attributes (pairs name – value). Non-leaf nodes are always directories while leaf nodes can be either directories or attributes. This structure is represented in the PEATS by two kinds of tuples. Directories are represented by tuples like (DIR, name, id, pid, ver) while attributes are represented by tuples like (ATT, name, value, id, pid, ver).

These tuples are used to represent all elements inside a namespace, and each element has an unique id field that is produced based on the id of the client that introduced the element concatenated with an internal sequence number of this client. The pid field of the tuples are used to specify which is the parent directory of the directory (in the case of a DIR tuple) or to which directory the attribute belongs (in the case of an ATT tuple).

Both tuple types have a version field ver which consists on a version number concatenated with the id of the client that inserted the tuple. This field is used to implement update operations on the NDS data, since a PEATS does not provide operations to modify tuples already in the space (more details in the next section).

2.2. NDS Service Algorithms

In this section we present the algorithms used to implement the basic operations of a LDAP-like NDS.

2.2.1. System initialization. When a process connects to a PEATS-based NDS system it reads the tuple of the root directory (a DIR tuple with pid = 0) and stores it on its root_dir variable.
2.2.2. Reaching a directory. Given an absolute name \( N = D_1/D_2/.../D_t \) the algorithm to reach a directory \( D_t \) is the following: beginning with the root dir, with \( id_r \), obtain the directory \( D_1 \) with \( pid = id_r \). If it does not exist then the absolute name does not exist in the namespace. Otherwise, repeat the algorithm with \( D_1 \) instead of root dir and \( D_2 \) instead of \( D_1 \) and so on until you reach \( D_t \).

2.2.3. Adding a directory or attribute. To add a new directory you simply have to put a new directory tuple in the space defined by its parent \( pid \). More precisely, to create a (sub-)directory \( D \) on a directory \( D_t \) with absolute name \( N = D_1/.../D_t \) a client has to execute the following steps:

1) Reach \( D_t \);
2) Obtain the \( id_t \) of \( D_t \);
3) Insert a DIR tuple in the space with name \( D \) and \( pid = id_t \).

The algorithm for creating the attributes of a directory is very similar and we do not present it due to lack of space.

2.2.4. Removing a directory or attribute. The algorithm to remove elements is also very similar to the one presented in the previous section: the first step is to reach the parent directory of the element to be removed and then remove it (storing its \( id \)). Additionally, if the removed element is a directory, all attributes with \( pid = id \) should also be removed. To be consistent with the LDAP specification, our algorithms do not remove directories with subdirectories.

2.2.5. Modifying an attribute. A common write operation on a NDS is the modification of the value of an attribute. More formally, to modify an attribute \( N = D_1/.../D_t/A \) to value \( v \), the following algorithm should be executed:

1) Reach \( D_t \) with \( id_t \);
2) Get all ATT tuples of attribute \( A \) with \( pid = id_t \);
3) Choose the tuple with highest version number \( ver \);
4) Insert \( \langle ATT, A, v, id_t, pid, successor(ver) \rangle \) in the space;
5) Remove all ATT tuples with \( name = A \), \( pid = id_t \) and \( ver < successor(ver) \).

2.2.6. Modifying a directory name. Renaming a directory given its old name \( D \) and new name \( D' \) requires the execution of an algorithm very similar to the one presented for modifying attribute values. The main difference here is that the directory name is changed instead of the attribute value.

1. \( successor(ver) \) denotes the next version number of a tuple with version number \( ver \).

2.3. Additional Aspects

Garbage collection and access control are two other important aspects that should be considered in our NDS implementation.

2.3.1. Garbage collection. The algorithms for modifying elements in the PEATS require the inclusion of a new version of a tuple before the removal of older versions. This approach may lead to problems if a client fails in the middle of a modifying operation (i.e., executes step 4 and fails before executing step 5 of the algorithm in Section 2.2.5). The algorithm works with many versions of the same element on the space, but this inconsistency may lead to poor performance. To avoid this, we foresee the use of a garbage collector (GC) that accesses the space sporadically and removes all tuples that represent outdated versions of elements. The algorithm is simple: the GC searches the whole namespace tree trying to find elements with outdated tuples and removing them.

2.3.2. Access Control. To avoid invalid accesses that can damage the tree structure of the NDS, we use the police-enforcement capabilities of the PEATS [2]. This control ensures that an operation is allowed to be executed on the NDS iff it does not impair the integrity of the tree structure. We omit further details due to space constraints.

3. Ongoing and Future Work

The algorithms presented here are being implemented as a JNDI (Java Naming and Directory Interface) driver that accesses the PEATS abstraction provided by the DepSpace [1]. The final objective of this work is to implement a LDAP-like NDS service that achieves almost the same performance of a BFT state machine replicated NDS.

Acknowledgment. This work was partially supported by FCT, through the Multiannual Funding and the CMU-Portugal Programs, and by the University of Lisboa – Fundação Amadeu Dias scholarship program.

References
