Automatic Reconfiguration for Large-Scale trustworthy Storage Systems

Abstract

Byzantine-fault-tolerant replication enhances the availability and reliability of Internet services that store critical state and preserve it despite attacks or software errors. However, existing Byzantine-fault-tolerant storage systems either assume a static set of replicas, or have limitations in how they handle reconfigurations (e.g., in terms of the scalability of the solutions or the consistency levels they provide). This can be problematic in long-lived, large-scale systems where system membership is likely to change during the system lifetime. This paper demonstrate the utility of this membership service by using it in a novel distributed hash table called dBQS that provides atomic semantics even across changes in replica sets. dBQS is interesting in its own right because its storage algorithms extend existing Byzantine quorum protocols to handle changes in the replica set, and because it differs from previous DHTs by providing Byzantine fault tolerance and offering strong semantics. This implements the membership service and dBQS.

Dr.B.G.Geetha,M.E.,Ph.D
Professor/Hod,
Department of computer science and engineering,
K.S.Rangasamy college of Technology,Tiruchengode-637215
E-mail:hodcse@gmail.com
Mobile no:9894688866

P.Senthil Raja,M.E.,(Ph.D)
Assistant professor,
Department of computer science and engineering,
K.S.Rangasamy college of Technology,Tiruchengode-637215
E-mail:visitsenthilraja@gmail.com
Mobile no:9994049209

R.Vijay sai,(M.E.)
Lecturer,
Department of computer science and engineering
K.S.Rangasamy college of Technology,Tiruchengode-637215
E-mail:visitsenthilraja@gmail.com
Mobile no:9994049209

Introduction

Today, Internet services become more important in functionality and store critical state. These services are often implemented on collections of machines residing at multiple geographic locations such as a set of corporate data centers. For example, Dynamo uses tens of thousands of servers located in many data centers around the world to build a storage back-end for Amazon's S3 storage service and its e-commerce platform. Additionally, these systems are long lived and need to continue to function even though the machines they run on break or are decommissioned. Thus, there is a need to replace failed nodes with new machines; also it is necessary to add machines to the

system for increased storage or throughput. Thus, the systems need to be reconfigured regularly so that they can continue to function. This paper provides a complete solution for reliable, automatic reconfiguration in distributed systems. This approach is unique because

- It provides the abstraction of a globally consistent view of the system membership. This abstraction simplifies the design of applications that use it, since it allows different nodes to agree on which servers are responsible for which subset of the service.
- It is designed to work at large scale, e.g., tens or hundreds of thousands of servers. Support for large scale is essential since systems today are already large and it can expect them to scale further.
- It is secure against Byzantine (arbitrary) faults. Handling Byzantine faults is important because it captures the kinds of complex failure modes that have been reported for this target deployments. For instance, a recent report about Amazon's S3 showed that a bit flip in a server's internal state caused it to send messages with the wrong content. Additionally, the Byzantine fault model makes it possible to tolerate malicious intrusions where an attacker gains control over a number of servers.

This solution has two parts. The first is a membership service (MS) that tracks and responds to membership changes. The MS works mostly automatically, and requires only minimal human intervention; this way it can reduce manual configuration errors, which are a major cause of disruption in computer systems. Periodically, the MS publishes a new system membership; in this way it provides a globally consistent view of the set of available servers. The choice of strong consistency makes it easier to implement applications, since it allows clients and servers to make consistent local decisions about which servers are currently responsible for which parts of the service. Using a small group for the MS is important since these protocols work well only in this case. The design provides scalability to a large number of nodes, most of which are clients of the MS. Additionally, it avoids overloading the servers that form the MS by offloading expensive tasks to other nodes. When there is a reconfiguration, the MS may need to move to a new group of servers. This allow the system to continue to operate correctly, even though the failure bound in the original group of MS replicas may subsequently be exceeded. A design for reconfiguring the Byzantine-fault-tolerant group is shown here. Tracking membership is only part of what is needed for automatic reconfiguration. In addition, applications need to respond to membership changes appropriately. Therefore, the second part of this solution addresses the problem of how to reconfigure applications automatically as system membership changes. This paper presents a storage system, dBQS that provides Byzantine-fault-tolerant replicated storage with strong consistency. This dBQS serves as an example application that uses the membership service and takes advantage of its strong consistency guarantees.

System model and assumptions

The system model and assumptions in entire project is shown here:A system comprised of nodes that can be servers implementing a storage service or clients using that service, without loss of

generality that the two sets are disjoint is assumed. By assuming nodes are connected by an unreliable asynchronous network like the Internet, where messages may be lost, corrupted, delayed, duplicated, or delivered out of order, this make no synchrony assumptions for the system to meet its safety guarantees, it is necessary to make partial synchrony assumptions for liveness. Assume the existence of the following cryptographic techniques that an adversary cannot subvert: a collision resistant hash function, a public key cryptography scheme, and forward-secure signing keys, also assume the existence of a proactive threshold signature protocol that guarantees that threshold signatures are unforgeable without knowing for more out of n secret shares. Assume a Byzantine failure model where faulty nodes may behave arbitrarily and compromised node remains compromised forever. This is a realistic assumption because once a node is Byzantine faulty, its secret information, including its private key, may be known, and therefore, it cannot be recovered, and then, continue to be trusted. Instead it needs a new key, which effectively means its identity has changed. These nodes have clocks whose rates should be loosely synchronized to keep time windows during which failure bounds must be met reasonably short.

The membership service

The MS describes membership changes by producing a configuration, which identifies the set of servers currently in the system, and sending it to all servers. To allow the configuration to be exchanged among nodes without possibility of forgery, the MS authenticates it using a signature that can be verified with a well-known public key. The MS produces configurations periodically rather than after every membership change. The system moves in a succession of time intervals called epochs, and this batch all configuration changes at the end of the epoch. Producing configurations periodically is a key design decision. It allows applications that use the MS to be optimized for long periods of stability (expect that in storage applications epochs could last for hours, although this evaluation shows that it can support short epochs if needed), and it reduces costs associated with propagating membership changes (like signing configurations or transmitting them). It also permits delayed response to failures, which is important for several reasons: to avoid unnecessary data movement due to temporary disconnections, to offer additional protection against denial of service attacks (assuming that this wait for longer than the duration of such attacks), and to avoid thrashing, where in trying to recover from a host failure the system over stresses the network, which itself may be mistaken for other host failures, causing a positive feedback cycle.

The notion of epochs provides consistency: all nodes in the same epoch see exactly the same system membership. Each epoch has a sequential epoch number. Epoch numbers allow nodes to compare the recency of different configurations. Furthermore, application messages include the epoch number of the sender; this allows nodes to learn quickly about more recent configurations.

The functionalities provided by MS are

- Membership Change Requests
- Probing

- Ending Epochs
- Freshness

Byzantine Fault Tolerance

To provide Byzantine fault tolerance for the MS, implement it with groups of replicas executing the PBFT state machine replication protocol. These MS replicas can run on server nodes, but the size of the MS group is small and independent of the system size. The MS operations are translated to request invocations on the PBFT group, and how to reconfigure the MS (e.g., to handle failures of the nodes that compose it).

PBFT Operations

PBFT provides a way to execute operations correctly even though up to f replicas out of 3 fare faulty. Therefore, these implements ADD and REMOVE as PBFT operations, which take as arguments the respective certificate, and whose effect is to update the current set of members (which is the PBFT service state maintained by the MS). Freshness challenges can also be implemented as PBFT operations.

Once the replica has collected the signatures, it invokes the EVICT operation, which runs as a normal PBFT operation. This operation has two parameters: the identifier of the node being evicted and a vector containing signatures from MS replicas agreeing to evict the node. The operation will fail if there are not enough signatures or they do not verify.

Reconfiguring the MS

There are two plausible ways to run the MS. The first is to use special, separate nodes that are located in particularly secure locations. The second is to use an "open" approach in which the MS runs on regular system members: servers occasionally act as MS replicas, in addition to running the application. This system can accommodate either view, by considering the first one as a special case of the open approach: This can mark servers (when they are added) to indicate the roles they are allowed to assume. At the end of the epoch, the system may decide to move the MS to a different set of servers. This can happen because one of the MS replicas fails; in the open approach it may also happen proactively (every k epochs) since the nodes running the MS are attractive targets for attack and this way it can limit the time during which such an attack can be launched. The steps that are needed to move the MS occur after the old MS executes the MOVEEPOCH operation.

Faulty Servers

Faulty servers cannot cause the system to behave incorrectly, it define in but they can degrade performance of this protocols. In the case of PBFT, this can be problematic due to the fact the primary replica plays a special role in the protocol, but recent work explains how to minimize this performance degradation. In this remaining protocols, the roles of different replicas are symmetric, so they are less affected by Byzantine replicas. The aggregation protocol is an exception: a single Byzantine replica can prevent freshness responses, but if this happens clients will switch to a different aggregator after a timeout.

Dynamic replication

The storage applications (or other services) can be extended to handle

reconfigurations using the membership service. In particular, it present dBQS, a read/write block storage system based on Byzantine quorums. dBQS uses input from the MS to determine when to reconfigure.dBQS is a strongly consistent distributed hash table (DHT) that provides two types of objects. Public-key objects are mutable and can be written by multiple clients, whereas content-hash objects are immutable: once created, a content hash object cannot change. In this paper, this describes only public-key objects. In a separate document, it describe the simpler protocols for content-hash objects, and also dBFT, a state machine replication system based on the PBFT algorithm, and a general methodology for how to transform static replication algorithms into algorithms that handle membership changes.

A complete, formal description of the main protocols used in dBQS and a proof of their correctness can be found in a technical report. Data objects in dBQS have version numbers that are used to determine data freshness, and identifiers that are chosen in a way that allows the data to be self-verifying (similarly to previous DHTs such as DHash). The ID of a public key object is a hash of the public key used to verify the integrity of the data. Each object is stored along with a signature that covers both the data and the version number. When a client fetches an object its integrity can be checked by verifying the signature using a public key that is also validated by the ID of the object. Version numbers are assigned by the writer (in such a way that distinct writers always pick distinct version numbers, e.g., by appending client IDs).dBQS ensures that concurrent accesses to the entire set of public-key objects are atomic: all system operations appear to execute in a sequential order that is consistent with the real-time order in which the operations actually execute. Designing the protocols for public-key objects from scratch is avoided, but instead extended existing Byzantine quorum protocols in novel ways to support reconfigurations, and provided some optimizations for them.

Performance during an Epoch

The performance of the MS during an epoch, and the impact of super imposing the service on dBQS servers are good. A more detailed evaluation of the base performance of dBQS can be found. Three types of membership activities happen during an epoch: processing of membership events, such as node additions and deletions; handling of freshness certificates; and probing of system members.

The main conclusion is that the architecture can scale well without interfering with dBQS performance

Moving to a New Epoch

The second part of the evaluation concerns the cost of moving from one epoch to the next. Here, this has two concerns: the cost at the MS and the impact on the performance of the storage protocols.

Cost at the MS

This part of the evaluation addresses the cost of reconfiguring the membership service at the end of an epoch. The deployed system in Planet Lab, which represents a challenging environment with frequently overloaded nodes separated by

a wide area network, and measured the time it takes to move the MS during epoch transitions. The goal of the experiment was to provide a conservative estimate of running the sequence of steps for changing epochs (PBFT operations, threshold signatures, and resharing). For analysis and evaluation of the individual steps. Ran the MS for several days and measured, for each reconfiguration, the amount of time that elapsed between the beginning of the reconfiguration and its end. The MS was running on a group of four replicas (i.e., fMS ¼ 1) and moved randomly among the system nodes. Even though, it is limited by the size of the test bed (in this case, hundreds of nodes), the main costs are proportional to the number of changes and the MS size, not the number of nodes, and therefore, this expect the results to apply to a larger deployment. Some reconfigurations were fast, but there is a large variation in the time to reconfigure; this is explained by the fact that nodes in the Planet Lab test bed are running many other applications with varying load, and this concurrent activity can affect the performance of the machines significantly. The main conclusion is that, even in a heterogeneous, often overloaded environment, most reconfigurations take under 20 seconds to complete. This indicates that the time to reconfigure is not a serious factor in deciding on epoch duration.

dBQS

When the system moves to a new epoch, any application that uses the MS must adapt to the membership changes. In this section, this evaluates the cost using dBQS. The experiment measures the cost of a reconfiguration by considering servers that run dBQS but are not implementing MS functions. In this case, the cost of reconfiguration is minor: for a particular client, there is the possibility that a single operation is delayed because of the need for either the client or the servers to upgrade to the new epoch, but all other operations complete normally. Note that this result assumes that state transfer is not required, which would be the case if the replica group did not move at the end of the epoch. State transfer can delay replies for objects that have not been transferred yet.

Conclusion

This paper presents a complete solution for building large scale, long-lived systems that must preserve critical state in spite of malicious attacks and Byzantine failures. By presenting a storage service with these characteristics called dBQS, and a membership service that is part of the overall system design, but can be reused by any Byzantine-fault tolerant large-scale system. The membership service tracks the current system membership in a way that is mostly automatic, to avoid human configuration errors. It is resilient to arbitrary faults of the nodes that implement it, and is reconfigurable, allowing us to change the set of nodes that implement the MS when old nodes fail, or periodically to avoid a targeted attack. When membership changes happen, the replicated service has work to do: responsibility must shift to the new replica group, and state transfer must take place from old replicas to new ones, yet the system must still provide the same semantics as in a static system. This show how it is accomplished in dBQS. This implements the membership service and dBQS. These experiments show that this approach is practical and could be used in

a real deployment: the MS can manage a very large number of servers, and reconfigurations have little impact on the performance of the replicated service.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels(2009)"Dynamo: Amazon's Highly Available Key-Value Store," Proc. 21st ACM Symp. Operating Systems Principles, pp. 205-220.

[2] J. Dean(2009) "Designs, Lessons and Advice from Building Large Distributed Systems," Proc. Third ACM SIGOPS Int'l Workshop Large Scale Distributed Systems and Middleware (LADIS '09), Keynote talk.

[3] Amazon S3 Availability Event (July 2008),http://status.aws.amazon.com/s320080720.html.

[4] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin(Apr,2009) "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," Proc. Sixth USENIX Symp. Networked Systems Design and Implementation (NSDI '09).

[5] J. Cowling, D.R.K. Ports, B. Liskov, R.A. Popa, (June,2009)"Census: Location Aware Membership Management for Large- Scale Distributed Systems," Proc. Ann. Technical Conf. (USENIX'09).

[6] D. Schultz, B. Liskov, and M. Liskov(Aug,2008) "Brief Announcement: Mobile Proactive Secret Sharing," Proc. 27th Ann. Symp. Principles of Distributed Computing (PODC '08).

[7] B. Liskov and R. Rodrigues, "Tolerating Byzantine Faulty Clients in a Quorum System(2006)" Proc. 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '06).

[8] J.R. Lorch, A. Adya, W.J. Bolosky, R.Chaiken, J.R.Douceur, and J. Howell,(2006) "The Smart Way to MigrateReplicated Stateful Services," Proc. European Conf. Computer Systems (EuroSys '06), pp. 103-115.