Reliable and Fault-Tolerant Peer-to-Peer Block Storage

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:23574653">http://nrs.harvard.edu/urn-3:HUL.InstRepos:23574653</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
Reliable and fault-tolerant peer-to-peer block storage

C. A. (Lex) Stein
Michael J. Tucker
and
Margo I. Seltzer

TR-04-02
Reliable and fault-tolerant peer-to-peer block storage

C. A. (Lex) Stein, Michael J. Tucker, and Margo I. Seltzer
Harvard University
Computer Science Technical Report 94-02 (TR-94-02)
{stein,mtucker,margo}@eecs.harvard.edu

Abstract

The Charles is a scalable, fault-tolerant, persistent block storage service built from a constantly changing network of potentially faulty or malicious computing nodes. The Charles is designed for a P2P environment: networks characterized by frequent process joins and departures, as well as arbitrary or Byzantine process behavior.

The Charles consists of two levels. The top level provides a scalable lookup service. This level is known as the Surface level and is structured as a ring of slowly changing process groups. The bottom level, known as the Submerged level, combines individual processes together into fault-tolerant groups absorbing network dynamics and process failure; whether it is arbitrary behavior or crash failure. The bottom level serves to provide the top level with predictable and well-behaved process groups. Both levels are self-organizing and self-managing. The Charles has no centralized control.

The Charles combines block lookup with storage. Fault-tolerance is provided through replication within process groups. Replication is maintained internally without any application state or callbacks.

1 Introduction

The Charles’ target computing base consists of heterogeneous machines, hardware, and operating systems connected by networks of varying latency and reliability. It is not expected that nodes will stay in the system for long periods of time. The Charles consists of processes contributed by physical machine nodes. Physical nodes with greater resources contribute more processes.

Over the past few years the P2P community has seen a number of attempts to adapt existing distributed systems algorithms to handle the distinctive properties of a P2P network. These networks are characterized by a large number of participating processes, short process lifetimes, and frequent joins and leaves. In addition, system administrators do not have access to the nodes and cannot control client operating systems or audit machine set-up to control behavior and security. These characteristics present challenges for the adaptation of existing work. Recent work on fault-tolerant distributed systems has sought to tolerate Byzantine or arbitrary process failure, but the resulting protocols do not scale well beyond a handful of nodes [1]. Recent work on scalable lookup protocols deals with process join and leave but has not addressed Byzantine failure modes [8] [6] [9] [5].

A recent paper by Sit and Morris [7] enumerated some of the security challenges facing peer-to-peer lookup services. In particular, those that threaten the liveness of the system. Broadly, these are attacks on routing correctness and attacks on data storage. Routing attacks include (i) incorrect lookup routing, (ii) incorrect updates, and (iii) network partitions. In most systems, the integrity of data relies on the collision resistance of the function used to map content to address. The liveness or availability of data is an issue not addressed by the addressing function. Data availability rests on replication. If all
potential data access paths pass through a single point, then access can be denied if that point fails or chooses to behave maliciously.

The Charles solves these security concerns using a two-level architecture. At the top level, or “Surface level”, individual process addresses are never used. Routing updates and lookups always operate on address vectors: collections of individual process addresses that are signed for verification. The Surface level ties fault-tolerant process groups together into a scalable ring structure. The Surface level has no global state. Each group maintains state regarding its neighbors. This state, taken together, provides the top-level ring integrity invariant: every group is reachable from every other group. In the worst case, with no address caching, a group would be reached by traversing along the ring from group to group successor. This structural invariant is similar to the one that Chord [8] seeks to provide, with the individual processes of Chord replaced with the Charles’ fault-tolerant groups of processes. The Surface level is composed of process groups. Groups do not fail suddenly and groups do not behave in arbitrary ways. Both crash and Byzantine failure are absorbed by the lower level or “Submerged level” protocols that govern the internals of the process groups. This level uses protocols for fault-tolerance that require multicast and group commit. These are known not to be arbitrarily scalable, and this is why they are submerged within the lower level of protocols.

The Surface provides scalability while the Submerged protocols provide resilience against many of the attacks outlined in [7]. These Submerged protocols do not intrude on the scalability of the Surface.

2 Related Work

Recent work in P2P substrates has taken routing protocols originally designed for static networks and modified these to accommodate joining and leaving processes. Most solutions have focused their attention on the issue of storing (key, value) pairs in a distributed system. Whether used to locate a service or a stored block, the basic problem is analogous to a distributed hash table.

Chord [8] modifies consistent hashing [3] to provide a distributed lookup service built on a 1-dimensional ring structure. Chord makes no liveness guarantees, relying upon higher-level systems to provide replication.

Chord is susceptible to network partitions created by malicious participants. In particular, a participant could route a lookup into a parallel, alternate ring structure populated by malicious cooperating processes. These participants might not have the data item being searched for. The client looking up the item has no way of knowing which process on the path diverted it into the alternate ring. The problem here is that there is no way of differentiating nodes of faulty or malicious processes from those of correct processes. The Charles’ address vectors solve this problem.

DHash [2] replicates blocks for the Cooperative File System [2]. Both CFS and DHash use Chord for lookup. Dhash is not resilient to failure because, for a given block, a single process manages replication.

CAN provides reliable hash table functionality that can withstand failstop and crash failures. Like the Charles, CAN allows multiple nodes to cover the same key space, and replicate the stored pairs. The semantics for joining are not Byzantine fault-tolerant. The CAN system depends on cooperation between nodes for handling unexpected enters and departures from the network, so a malicious node could potentially force the routing tables maintained by neighbor nodes to become inconsistent [3]. CAN proposes the use of multiple hash functions for replication to avoid reliance on a single process [5].

Tapestry [9] and Pastry [6] modify Plaxton routing [4]. While these lookup systems are designed to deal with process crashes and the arrival of new processes, neither deals with malicious or Byzantine nodes. And, like the consistent hashing systems, malicious nodes can disrupt correct operation.

The Byzantine file system (BFS) [1] implements a Byzantine fault tolerant NFS server. However, this system does not scale well beyond
a handful of nodes.

One of the goals of our research with the Charles is to discover if Byzantine fault-tolerant algorithms can perform as well as less fault-tolerant alternatives during periods of stability and better during periods of high instability. Broadly, concept of stability includes both the correct behavior of processes and the rate at which processes join and leave the system.

3 The Surface Level

Charles’ blocks are addressed by the secure hash of their contents. Groups are described by the interval of the Charles’ address space they span. Individual processes are not assigned to specific Charles’ addresses. However, the hash of their IP address and virtual tag specifies which group they join. A process may be responsible for any blocks that map into the address interval of its group.

The Surface level of the Charles consists of a ring of process groups. Groups’ intervals do not overlap and the union of all group intervals at any time covers the Charles’ address space. Intervals change only during group merge and split. Charles’ clients and members of the Charles use address vectors (AVs) to address process groups. AVs have several components:

1. The Charles’ address space interval spanned by the group. All processes in a group agree on this interval.

2. A numeric generation number.

3. A list of process addressing information consisting of the host IP address, virtual tag, and public key.

The generation number is monotonically increasing across group merges and splits (known as structural group changes). When a group splits, the two resulting groups will increment their generation number. When two groups merge, the generation number of the single consequent group is set to the increment of the maximum of the two generation numbers. Different groups may have the same generation number. Generation numbers are invariant across process joins and departures. Generation numbers are used to generate merge request certificates that are resilient to replay attack and to invalidate old AVs.

The virtual tag limits the number of processes that can be contributed by a given host at an IP address. Each IP address has \( k \) possible virtual tags. These are generated from the IP address by \( k \) different hash functions. The virtual tag exists to accommodate resource differences across machines. A machine with surplus storage can contribute many processes while a machine with scarce storage can contribute a small number of processes. The virtual tag distributes a host’s processes across the Charles’ address space, ensuring that, in a reasonably sized network, a malicious host will not be able to target groups and overwhelm them with its processes.

3.1 Address Vectors

Every process in a group possesses two AVs; one pointing to its succeeding group and another pointing to its preceding group.

AV authentication is inductive. That is, once a client has a trusted AV, it can use this to trust another AV. Bootstrapping trust is a challenging problem and left for future work. Currently, we use the same technique as everyone else: relying on an external third-party for bootstrapping trust. Examples of these sources are well-known web addresses, or a trusted government body or corporation.

We now outline how a client can use a trusted AV to transfer trust to a second AV. Client \( C \) possesses an AV for group \( G_1 \) that it trusts. \( C \) wants to walk the ring to find a document and asks \( G_1 \) for \( G_1 \)'s succeeding group. A process of \( G_1 \) responds with a view (in the form of an AV) of its successor signed by \( f + 1 \) members of \( G_1 \). \( C \) verifies these signatures. Since at most \( f \) members of \( G_1 \) can be faulty, if \( f + 1 \) of \( G_1 \) agree on the AV of \( G_2 \), then it must be correct. \( C \) now trusts the AV for \( G_2 \), which contains the membership list of \( G_2 \).
AVs do not fail like nodes or processes. Their correctness depreciates as processes leave and enter the group. An AV that is completely correct at time $t$ will be less correct at time $t + 1$, when process $N_X$ joins, and even less correct at time $t + 2$, when process $N_Y$ leaves. Consequently, AVs must be refreshed periodically; the new AV must be authenticated. This is done using the current AV. The current AV contains a mapping from process to public key for at least $3f + 1$ processes. At most $f$ of these can fail or leave the group during a given time period. Therefore, at least $2f + 1$ must still be correct. The new AV is authenticated by verifying at least $f + 1$ of the signatures appended to the end of the new AV using the public keys from the original AV. If these signatures are valid, the new AV can replace the old one since $f + 1$ agreed and we know that at most $f$ processes are faulty.

4 The Submerged Level: Process Group Protocols

Process groups isolate the Surface level from Byzantine failure and temper network dynamics. Group protocols are designed to tolerate at most $f$ failed processes during a time period of length $t$.

The number of processes in a group varies between a low-water mark $L$ and a high-water mark $H$. When a group has more than $H$ processes, one of its members will initiate the split protocol. A split partitions the processes into two new groups, each approximately one half the size of the initial group. When a group has fewer than $L$ processes, one of its members will initiate the merge protocol. The merge combines a group with its neighboring group. If the resulting group has greater than $H$ processes it will begin a split protocol that will evenly divide the nodes.

Both large and small groups can only tolerate the same number of process failures. Consequently, larger groups can tolerate a smaller percentage of failing processes.

4.1 Membership protocol

Each process maintains a vector of processes that it has heard from. This is called the heard-from vector.

Every time a process receives either a “join” or “heard-from” message from a process it adds the process to its heard-from vector.

A process receives heard-from vectors from other processes and adds these into a column of heard-from vectors. If $f + 1$ vectors agree that they have heard from a process, then this process is added into the send-to vector. The initial send-to vector consists of the process’s heard-from vector.

Every time period $t$ a process sends its heard-from vector to everyone in its send-to vector. If a process hasn’t heard from a process in $m$ time periods, then it removes the process from its heard from list.

4.2 Process Join

A joining process starts with any AV representing a group within the Charles network it wishes to join. This AV must be obtained from a trusted source, such as http://www.the-charles.net. The process multicasts a signed join message to the processes listed in the AV. This message contains the joining process’s IP address, virtual tag, public key, and a random cookie to the processes listed in the AV. The joining process then waits for $2f + 1$ verified acknowledgments and then considers itself a member of the group. The joined process constructs its initial heard-from list from the $2f + 1$ acknowledgments and participates in the membership protocol.

4.3 Process Crash Failure

If a process doesn’t hear from a process within $m$ time periods, then the latter is removed from the former’s heard-from vector.

4.4 Group Split

A group’s Charles’ address space coverage changes across merges and splits. This group
interval does not change across process joins and failures.

When a process has more than $H$ processes in its heard-from vector, it will initiate the split protocol by multicasting a “split desired” message to members of its send-to vector. If a process has more than $H - d$ (where $d$ is a small positive number) in its heard-from vector and receives a “wanting to split” message it will reply with an “OK” message and multicast a “wanting to split” message of its own to all the members of its send to vector with the exception of the process that it just replied to. Once a process has multicasted a “wanting to split”, it waits to receive $2f + 1$ “OK” messages. Once the process has received the $2f + 1$ verified “OK” messages, it moves to the pre-split state and begins a Byzantine fault-tolerant distributed consensus algorithm to determine the pivot address that will split the group in two. Once this address has been chosen, all processes are able to construct their new view of their group and the new AV of one of their two external AVs that points to the newly formed group drawn from their former groupmates. The preceding and succeeding groups are informed of the new AVs.

4.5 Group Merge

The merge protocol is initiated by a process when it has less than $L$ processes in its heard-from vector. The initiating process sends a “merge desired” request to every process in its send-to vector and waits to receive “OK” message replies. If a process has fewer than $L + d$ processes in its heard-from vector and receives the “wanting to merge” message, then it will respond with an “OK” message and multicast a “wanting to merge” message to all the processes in its send-to vector. Once a process has sent out a “wanting to merge” message it waits for $2f + 1$ “OK” replies.

If a process does not receive $2f + 1$ messages within a time period the process times out and expires. The physical node hosting the dead process may elect to continue allocating its resources to the Charles by spawning a new process. The new process will go through the join protocol just as any new process does. It may be assigned a different hash function and consequently the node’s resources would be allocated to a different group.

Once a process receives $2f + 1$ “OK” replies it knows that at least $f + 1$ of these must be non-faulty. Each process sends its compiled merge certificate to all the members of its successor group. The merge certificate consists of all $2f + 1$ signed “OK”s. This is enough to convince the successor group processes that the group legitimately wishes to join.

A successor group cannot refuse a join. The merging group and its successor both know their own and each others generation numbers and Charles’ address intervals. Knowing these, they need no communication to compute what the generation number and interval of the single merged group are. The new generation number is the increment of the maximum of the two numbers. The new Charles’ address interval is the union of the two intervals.

4.6 Block Storage and Replication

Clients must write at least $f + 1$ copies to the group to ensure that one copy has been committed on a non-faulty process that begins the internal replication. Clients can proceed asynchronously, in parallel with the commit of the internal group replication or clients can wait synchronously for confirmation that a block has been successfully replicated across the group.

On group splits, storage will be freed across the splitting processes. On group merges, blocks will be copied across to the processes that do not already store them, decreasing storage capacity.

5 Conclusion

The Charles addresses the security concerns of data and routing liveness by ensuring that addresses used for lookup and routing only point to correctly behaving entities. This is achieved with a two level system. The top level provides scalable routing across correctly behaving groups.
while the bottom level hides the protocols that guarantee group correctness to the top level.

Replication is managed within Charles process groups and avoids the single point of failure that has exposed other systems to potential attack.

We are in the early stages of the Charles' development. Future work will measure performance against other peer-to-peer substrates and measure the costs associated with the Charles' liveness guarantees – both the fault tolerance of routing and data replication. In addition, we plan to further explore and understand the sensitivity of the system's performance and fault-tolerance to its parameters.

References


