Abstract

Applications in the realm of The Internet of Things often depend on the cooperation between devices with different resources at their disposition. For example, devices in this kind of heterogeneous networks may have exclusive access to large disks, powerful CPUs and GPUs, sensors, actuators, etc. Additionally, IoT networks are often subject to constant node failures, unreliable network channels and network partitions. Distributed applications looking to take maximum advantage of the features of each node require precise control of the data flow throughout the network while preserving fault-tolerance and availability. Alternator provides a fault-tolerant, highly available solution to the problem state-sharing for distributed applications in heterogeneous networks. The Alternator key-value store achieves this by using a consistent-hashing scheme, much like Chord [4], as a bottom layer to store metadata for each entry in the system. In order to provide full control of the data flow in the system Alternator allows users to specify the nodes in the system that will replicate each entry. This offers complete granular control of data-flow in the system while also preserving all of the virtues of consistent-hashing schemes: speed and fault-tolerance.

1 Introduction

Internet of Things networks are typically composed of a heterogeneous set of devices, each one with its own set of unique features and resources at its disposition. These characteristics pose many challenges to applications that are distributed across modern IOT networks, since maximizing performance requires a precise control of the data flow throughout the entire system. For example in order to achieve optimal performance, an application may need to direct essential records to the node with the most reliable hard disk, or push all vector data to the node with the most powerful GPU.

These types of networks are also fault-prone, often-times due to hostile environments, limited power sources or unreliable network links. This further complicates the task of any distributed application, as failing nodes and connections may delay or interrupt the data flow.

Previous solutions to the state-sharing problem typically rely on the idea of consistent hashing to distribute data evenly throughout a network. Decentralization, even distribution of load and speed are the greatest virtues of this scheme, which are all well demonstrated by the Chord [4] system. Many other systems have brought these features to more specific settings, such as the low-powered networks in FAWN [1] or large content distribution networks in the case of Amazon’s Dynamo [2]. The pitfall of consistent hashing is that an even load balance implies homogeneity. For example, in a basic implementation of Chord with a reasonably large amount of nodes, keys in the system are spread out almost evenly throughout the nodes. Under the assumption that applications access keys evenly throughout the key-space, then the read and write load on the system will be distributed evenly throughout the nodes as well. It is indeed possible to fine tune the load put on each node by filling the ring with virtual nodes instead of the actual nodes themselves. Unfortunately, the idea of virtual nodes only allows the system to balance load by the raw throughput of each node and not other unique features that members of the network may possess. The approach to fault-tolerance in ring-based systems has been more successful, with schemes such as chain-replication in FAWN and a quorum-based mechanism in Dynamo.

Alternator demonstrates how the concept of consistent hashing can be used as a layer in order to bring flexibility in data flow control while maintaining many of the useful properties of ring-based systems. The key-value store implemented in Alternator gives full control of data location within the network to the user: for each key-value pair entered in the system the user can also specify the nodes that will store this entry. Alternator then takes care of the mapping of keys to nodes specified initially, so that future queries do not require any knowledge of where data is stored. This metadata is stored on Alternator’s base layer: a consistent hashing ring similar to the one described in the Chord paper. This layer implies additional work for queries, since now an additional network jump is required to find the metadata associated with a node. This expense is made so that Alternator can offer granular control of the key distribution.

Fault-tolerance and availability are also of high impor-
tance to Alternator. Although any Get operation will require that all of the nodes storing a key are available, node failures have the potential to slow down queries to the system if nodes storing the metadata needed for the query are unavailable. Alternator’s architecture allows for the same fault-tolerant mechanisms implemented in FAWN and Dynamo, and for now a more FAWN-like approach has been taken.

2 Architecture

Alternator supports two main operations: Get and Put. A Get operation is a query to receive the value associated with an entry in the store. A Put operation is a query to insert an entry into the store. In order to achieve this, a node in Alternator stores two types of information: data and metadata:

Data is simply the value in a key-value pair as entered by users into Alternator. Note, however, that users are not expected to directly handle the keys in key-value pairs. Instead, the user is meant to use a name to refer to each entry in the database, just like variables in a programming language. Alternator then hashes the name and uses it as the the primary identifier of the entry throughout the system.

Metadata contains information on a specific key-value pair in the system: the original name of the key-value pair as used by the user to Put and Get its value, and a list of nodes that store the the value of the pair. This list of nodes is called the list of replicators.

Nodes in Alternator arrange themselves into a ring. Each node in the system is assigned a unique identifier that belongs to the same key-space as entries in the system. This ring has a circular order, where members are ordered lexicographically by this primary identifier and the successor of the node with the highest identifier is the node with the lowest identifier. Each node is primarily responsible for storing the metadata of the set of all pairs for which it is the immediate successor of the pair’s key. The successor of a given key is called the coordinator of that key.

Thus, any Get or Put operation looking for the value \( v \) of a key \( k \) consists of two steps:

1. Contact the successor of \( k \), and either acquire the metadata (Get) or insert it (Put).
2. Get \( v \) from any of the replicators (Get) or insert \( v \) into each (Put).

Data replication and fault-tolerance are tasks left entirely as a responsibility of the user. On each Put operation the user must specify the nodes in which they desire to store their data, the list of replicators. The only way in which Alternator ensures data availability is that upon node departures the departing node ensures that all of its data is stored in one more node, in order to preserve the original amount of replicators. Under conditions in which nodes are unavailable during a Get operation Alternator queries replicators until an available one is found. Note, however, that given the power of full data-flow control it is possible for an application to intelligently distribute its data in the network as to maximize availability and fault-tolerance given information on the availability of each node. This is the problem that is being currently tackled by Soham Sankaran[3]. Alternator, when combined with one such tool capable of doing this optimization, could provide a very powerful tool to applications operating on large amounts of data in an IoT network.

Metadata replication, on the other hand, is handled by the system, promising high availability even during situations in which many nodes in the system are unavailable. This guarantees that data in the system can be found in constant time under duress. Replication of metadata is done by storing each piece of metadata in the system both at the coordinator and its (N-1) successors. This means that each metadata entry is replicated in \( N \) different nodes. Only if these \( N \) nodes fail will the system fail to find the metadata. The value of \( N \) is available to the user as a configuration option.

2.1 Key-value store operations

2.1.1 Get

A Get operation acquires data from Alternator. It has simply one parameter, the name of the entry. Its result is the value in the entry. Any node in Alternator can fulfill any Get request, but depending on whether the node is the successor of the key being requested the node may have to forward the request to another node, preferably the key’s coordinator. If the coordinator is down, then linear probing is performed on its N-1 successors, stopping at the first available one. The first available node in the chain can then acquire the metadata for the key from its own database, find the list of replicators, acquire the data from any of the replicators and forward it to the original source of the request.

Get operations are reliable even during constant membership changes: if the original receiver of the Get request is unaware of a recent change in membership which changed the coordinator of the key then the request will get forwarded to the incorrect node. But even if forwarded incorrectly, the new receiver of the request may still forward it. Since node departures and joins are handled by the successor of the joiner or leaver the Get request will eventually reach the joiner/leaver’s successor,
which will be aware of the change in membership and will forward the request correctly.

Get requests are also fault-tolerant, thanks to Alternator’s metadata replication scheme. When a node fails to forward the request to the key’s coordinator due to an outage an attempt is made to forward the request to one of the $N$ successors of the key. Assuming that any of the $N$ successors are available, then the query for metadata in the Get request will be successful in less than $N$ network operations. A Get will only fail if all of the $N$ successors or all of the replicators are unavailable.

### 2.1.2 Put

A Put operation is simpler. The handler of the operation is found in the same way as the handler of a Get by looking for the first available node in the metadata chain. Every other member of the replication chain is then given the metadata of the new entry, and each replicator is given a copy of the data. Alternator features a fail-safe mechanism to ensure that a Put operation fails if a significant portion of the chain or the list of replicators fail to confirm that they have successfully stored their corresponding data or metadata. This mechanism may be necessary due to the possibility that the rest of the chain or replicators also fail before the rest become available, which could lead to the loss of information. Currently, half of the metadata chain or $k$ of the replicators must fail for the mechanism to activate, where $k$ is a user-specified parameter. If either the chain or the replicators fail to meet these criteria, then the Put is canceled, the chain and replicators are indicated to drop the received data and metadata, and an error is returned to the user.

Furthermore, if the fail-safe mechanism is not activated but part of the chain or some of the replicators failed to confirm a successful operation, then the handler of the operation records these events in a local database of pending operations. All of the operations in this database are then reattempted periodically, in an attempt to resolve them all. This time period is a user configuration option.

### 2.2 Ring membership

Alternator implements a ring structure in order to store any key-value pair’s metadata. Each node maintains a full membership list of the ring, allowing queries to be resolved in constant time. This design decision is affordable because each entry in the membership list occupies a very small amount of space, and IoT networks are typically not large enough to warrant the distribution of the membership across nodes.

Changes to the membership list are assumed to be explicit, and so manual intervention is needed in order to add or remove nodes from the system. Therefore, the assumption that all node departures are graceful is made. However, this does not imply that the system is not fault-tolerant: unresponsive nodes are simply expected to eventually become available again and leave gracefully. Also, node outages do not imply the temporary unavailability of data if the data is replicated in more than one available node, and metadata will be available unless the entire chain of $N$ nodes fails.

Graceful leaves are not a strong requirement of the system, but they are assumed for implementation simplicity: It would be possible to implement a protocol so that available nodes in the system can agree to remove an unavailable node from the system, so that Puts and Gets are no longer forwarded to that node. After a node is kicked, then members of all the chains it belongs to could ensure to restore the size of the chain to $N$ by adding a new member, and replicators of any data that was held by the kicked member could add another node to the set of replicators.

The membership list is built from a history of membership changes throughout the system. Each node builds their membership list from their history and history changes are propagated using a simple gossip protocol. This protocol is similar to the anti-entropy techniques used in more complex gossip systems: Each node regularly compares their own membership history with the history of a random node, incorporating all entries that are new to the node and subsequently rebuilding their membership list if new entries were found. This allows membership changes to be propagated in approximately logarithmic time throughout the network, as shown by the Watts and Strogatz model [5].

Entries in the membership history contain a single membership event concerning a single node. Each entry has three fields: 1. the type of entry, either leave or join, 2. the identifier and network address of the node joining or leaving, 3. the time at which the change occurred. This information is sufficient to create a membership list by simply following the history linearly with time. The main disadvantage of this scheme is that the clocks for nodes have to be synchronized so that the order of membership changes is the same as the order in which changes were made in real time. Nonetheless, since changes in the membership must be made explicitly by administrators then the clocks of nodes do not have to be unreasonably synchronized. Unless possibly conflicting changes (involving the same nodes) are made by different administrators in different nodes at the same time, conflicts or ordering errors are extremely unlikely as long as the clocks are synchronized in the order of seconds, which is a solved problem for operating systems that synchronize their clocks with a network server.
2.3 Membership Changes

2.3.1 Node joins

A node can join an already existing ring by establishing a connection with any other node already in the ring. The new node simply queries the proxy node to find its successor. The successor of the node attempting to join the ring is called the broker. Upon finding the right broker, the new node initiates an operation called a Join Request. During the request the broker must fulfill two tasks in order to maintain the ring’s properties: First, the broker must ensure that the new node acquires all of the metadata that corresponds to it, both as coordinator and as a member of possible metadata chains. In a stable state, the broker will have copies of all of these entries, and so it simply must query its own database for the range corresponding to the new node and pass them on as a response to the request. The range of entries that the new node must store is simply [ID of Nth predecessor of the broker, ID of broker]. Secondly, the broker appends the addition of the new node to its own history. The synchronization mechanism will then ensure that other nodes find out about the node’s addition to the ring. Upon receiving a successful response for its Join Request, the new node must add all of the entries it received from the broker to its database. Finally, the new node initializes its membership history by copying the broker’s history after the broker has already added the new node.

2.3.2 Node departures

A graceful node leave requires the assistance of the leaver’s successor, so that the leaver can ensure that all of the data and metadata it owns is not lost permanently. The leaver carries out two operations: First, it gets all of the metadata entries that it coordinates and sends them to its successor, which will be their new coordinator after the departure is complete. Second, it ensures that all of the data it is in charge of replicating is also replicated in some other node. This mechanism is a very primitive implementation in the current version: it simply goes over each data entry and performs an operation called RePut on each entry. RePut simply modifies the metadata in the ring to remove the leaver from the list of replicators, adds some other random node in the ring to be an additional replicator and sends the data to this new random node.

2.4 Fault tolerance

The fault tolerance of Alternator comes from its replication scheme. As has been mentioned before, Alternator is not entirely tolerant to loss of data due to node outages. This is entirely dependent on the amount of nodes in which the user decides to put each entry on a specific Put. If during its initial Put an entry is only replicated in one node, then if that node fails the entry will be lost as long as the node remains unresponsive. However, if the entry was replicated in several nodes, then as long as one remains responsive the entry will be accessible. This problem could be solved in a further version of Alternator, where if an outage is detected then nodes replicating data in the failed node can ensure that one more copy is created in order to maintain the original amount of replicators. A simple version of this is already implemented: when a node leaves gracefully then it ensures to replicate its data in one more node.

Although the current Alternator implementation is not entirely tolerant to the loss of data it is highly tolerant to the loss of metadata, even during conditions of constant node outages. This ensures that as long as the nodes replicating an entry in the database are live, the user can find and access these keys. Nonetheless, even under this condition a linear search can be performed through the ring in order to find the key-value pair, so data is never lost due to the unavailability of metadata, only the query is slowed down (linear searching is left up to the user, and is not performed by Alternator in its current version).

3 Implementation

All of the features discussed in this paper have been fully implemented, unless specified otherwise. The entire implementation was done in the Go programming language. Go was chosen for its outstanding support of concurrent operations. Additionally, the Go compiler produces native machine code and is capable of compiling to several different architectures. The current Alternator implementation consists of three packages. The main package is a Go library, which can be used by other applications to initialize Alternator nodes, as well as interact with them. An application designed to interact with Alternator can use this library to make Get and Put operations on an Alternator system, using the same remote procedure call mechanism used by Alternator nodes. The second package is an executable that initializes an Alternator node and takes command line arguments for configuration. The third package is a testing suite that initializes a set of nodes into a ring and allows the user to insert keys at random, add nodes to the ring, get all the keys in the system (and verify their correctness) and dump all data and metadata in the nodes.

Get and Put operations in Alternator’s key value store are meant to be made using a name as an entry’s primary identifier, much like variables in a programming language environment. The internal identifier of a key is the 20 byte SHA1 hash of the key’s name. This provides a large enough key-space so that collisions between node
identifiers (also in this key-space) are highly rare. An external Go implementation of a key-value store database is used as a node’s primary database: Bolt (which will be replaced in the future, as stated in the performance section).

Communication between nodes is done through Go’s RPC (Remote Procedure Call) package (net/rpc), which allows any node in Alternator to call functions in other nodes, making the code highly readable and straightforward. The RPC package is highly flexible in its use of different protocols to make remote procedure calls, and it is simple and straightforward to initiate communication with other nodes through different channels. During development Alternator was strictly tested with communication over TCP, using HTTP as a top layer. Adapting the current implementation to work over other communication mechanisms is a straightforward process, and the code was built with this possibility in mind. This is important given the variety of communication standards in modern IOT networks, such as WLAN and Bluetooth.

4 Results

4.1 Performance

Performance has proven to be Alternator’s main challenge. In its current implementation, Alternator is still very slow and it is very easy to flood the ring with Put requests. This is due to sub-optimal database management and RPC connections, as evidenced by tests in which database operations where replaced with empty operations.

Currently, Alternator nodes operate on the disk to serve every single Get and Put request, as the Bolt database does not provide an in-memory cache. This problem can be easily solved by changing the database solution used by nodes, ideally one with an in-memory cache. An in-memory cache would guarantee that Put operations are faster than a disk operation, which has a large overhead both due to disk and file-system limitations. The in-memory cache would ideally be committed to disk when the node is not serving a large amount of requests, although it is also important to consider the possibility of a memory failure and information stored in memory being lost. Additionally, Bolt seems to not be very efficient at batching together operations performed from different routines (a routine is a lightweight Go thread). Batching operations is highly important so that operations running in different routines do not constantly block each other by acquiring the database lock. Bolt was proven to do a poor job of batching operations by replacing all of Bolt’s put operations with a mock database write (wait for 1 millisecond) and a simple lock, simulating a more efficient database operation. This lead to an improvement of one order of magnitude.

Figure 1: Time of 1000 Put operations under no traffic conditions (one request per second). Tested on a 16 node ring where $N$ is 3 and each operation inserts a random string of 30 characters in any number of nodes.

Alternator’s performance is also hindered by the current RPC library being used, the official Go net/rpc package. Unfortunately, net/rpc makes an extreme use of bandwidth. During bursts of Put operations the network gets severely strained and some of the remote procedure calls fail or timeout. These fails and timeouts are extremely taxing on the system, since they imply more database operations, as nodes have to record these incidents on their database in order to resolve the pending operations. This strains the database further, which causes more operations to block due to the Bolt’s inefficient batching.

Nonetheless, this performance problem is not a dead-end. Bolt can be replaced with a better database option with an in-memory cache, which would help improve performance during large bursts of operations. The only requirement is that the replacement is also compatible with a variety of architectures (which Bolt achieves by being a Go library). No specific alternative has been chosen yet, and given the simplicity of the problem it would be possible to implement the local key-value store as part of Alternator instead of depending on third party libraries. Furthermore, the current RPC library could be replaced with one of the many better performing ones available for Go, although no specific decision has been made.
Figure 2: Comparison of sets of 1000 Puts with different time intervals between puts, showing the performance degradation of Alternator due to traffic. Tested on a 16 node ring where $N$ is 3 and each operation inserts a random string of 30 characters in one node.

4.2 IoT Devices

Other than Go’s great facilities for concurrent code, Go was also selected as a tool for this project due to its good support of a large variety of architectures. It is a one-step process to compile Go code for non-x86 systems, and cross-compilation is a non-issue. Alternator has been cross-compiled to the ARMv6 architecture, and rings have been tested where some of the nodes are Raspberry Pi devices. Performance was not thoroughly tested in these devices, but given that current performance is highly limited by software capabilities, it is unlikely that it differs in Raspberry Pi or similar devices. When software performance is improved thorough testing in low-powered devices will follow.

5 Future Developments

Alternator does not intend to be a theoretical model for an IoT key-value store. Instead, Alternator attempts to be a working solution for the problem of state-sharing within the realm of IoT. Alternator is still far from this goal, and although it lays down all of the theoretical concepts in this document and the current implementation is 2300 lines of code, it is still far from being a complete solution. A lot of work still has to be done in making Alternator more fault-tolerant, robust and faster. Alternator’s current fault-tolerance mechanisms become weak under heavy-stress situations, such as when the system is currently handling incoming Put operations while recent membership changes have not yet stabilized. The handling of these scenarios is critical for Alternator’s usability. The solution to this kind of problems is not particularly complex but it is painstaking to implement. Simply, nodes require a mechanism to ensure that all of the entries they coordinate are properly replicated. The naive solution to this problem has nodes compare their own databases with its successors, ensuring that metadata is replicated properly, but this operation is very costly. A more sophisticated solution has nodes keep a hash of the contents of their databases within a given range, then this hash could be used to quickly make the comparison. The costly operation of comparing sections of the database would only become necessary when the hashes do not match. This can be extended to the creation of Merkle trees, so that finding mismatches between databases becomes an $O(\log n)$ operation. Once this mechanism is implemented, the other greatest lacking of Alternator can be addressed: Currently, Alternator has no mechanism to evict entries from nodes that should no longer be storing them. But assuming that coordinators of entries can ensure that replication is correct, then coordinators could initiate the eviction of entries in other nodes safely. This will ensure that Alternator nodes do not accumulate garbage as they do presently.

As shown in several other systems, consistent hashing by itself is not good enough to ensure an even load distribution in large networks. This means that requests involving metadata may still flood the least capable devices in Alternator if they happen to coordinate the metadata of a several important keys. The solution to this problem is having each real node in the system have a number virtual nodes representing it in the ring, as done in FAWN [1] and Amazon’s Dynamo [2]. The implementation of virtual nodes in Alternator will alleviate this problem.

6 Conclusions

This document describes the design of all the features that Alternator requires to become a complete solution to the problem of state sharing and data flow control in fault-prone heterogeneous networks. The current implementation is still very rough, and is still more of a proof of concept than a working solution. Furthermore, its performance is still a great limitation, some features are very roughly implemented, and the fully implemented features can still be further optimized.

Nonetheless, Alternator promises to be a fully working system with a large and increasing set of possibilities. It
may one day become a state-sharing solution for large sensor networks in hostile environments or independent robot swarms.

Acknowledgments

Special thanks to Professor Mahesh Balakrishnan for being my advisor and Soham Sankaran for trusting my project as a dependency for his CPSC 625 project.

References


