AVAILABLE GROUP KEY MANAGEMENT FOR NASPINET

BY

JASHUA GUPTA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Peter W. Sauer

ABSTRACT

The North American SynchroPhasor Initiative Network (NASPInet) aims to provide a secure and reliable communication infrastructure for sharing phasor measurement unit (PMU) data among utilities, balancing authorities, reliability coordinators and other power grid entities. Since the data is secured using cryptographic primitives such as encryption and message authentication codes, it is critical to ensure fault tolerance of the key management infrastructure that the cryptographic primitives rely on. Since an entity might be sharing its data with multiple other power grid entities, group cryptographic primitives are more efficient and thus the keys must be distributed and managed among a group of authorized entities. To make group key management (GKM) fault tolerant, we must eliminate its single point of failure – the key distributor (KD). To ensure that the system can keep running even if the KD experiences failure, we maintain its replicas, which can replace it whenever needed. If not managed efficiently, replication can lead to problems like lack of consistency between replicas, deadlocks, and decreased throughput of operations. Hence, it is necessary to coordinate operations. Instead of performing coordination from scratch, it is much more efficient to make use of a coordination service that has proven to be successful in doing so for other applications. In this thesis, we propose the design of an interface that enables the group domain of interpretation (GDOI) group key management protocol to replicate its KD, the group controller/key server (GCKS), by using the ZooKeeper open source coordination service.

CONTENTS

1. INTRODUCTION

Wide-area situational awareness (WASA) has been recognized as a key enabling functionality for smart grids. Key enablers of WASA are the time synchronized and precise grid measurements called synchrophasor measurements. These measurements are made possible by phasor measurements units (PMUs), GPS clock synchronized measurement devices capable of measuring the current and voltage phasors in the power grid. Recognizing the potential of enhanced data collection by PMUs, the U.S. Department of Energy (DOE), North American Electric Reliability Corporation (NERC), electric utilities, federal and private researchers, academics, and others have started a collaborative effort, the North American SynchroPhasor Initiative (NASPI) [1].

One of NASPI's aims is to provide a data communications infrastructure called the NASPI network (NASPInet) which enables utilities or control centers to share the data collected from hundreds of thousands of PMUs across the grid. PMU data consists of voltage and current phase angles, and rate of change of frequency, and is sampled 30-120 times per second. Hence, PMUs give much more direct access to the state of the grid than conventional SCADA systems which sample data on voltage and current magnitudes every 2-4 seconds. The PMU data is sent to phasor data concentrators (PDCs) at substations or control centers to be used for synchrophasor applications. The data then may be shared by the control center with its reliability coordinator and other control centers to enable WASA. NASPInet) aims to enable secure sharing of PMU data.

The proposed architecture of NASPInet [1, 2], as shown in Fig.1, was designed keeping in mind the importance of wide-area sharing of PMU data. NASPInet will be composed of phasor gateways (PGWs), which shall be the sole access point of entities like utilities and monitoring centers, and a data bus (DB) which includes a wide area network (WAN). The DB enables the PMU data to be shared among hundreds of PGWs. The data is used for applications ranging from feedback control to situational awareness. It is, therefore, of prime importance to secure the confidentiality, integrity, and availability of PMU data to ensure the reliability of applications that use it and hence that of the power grid. We give a brief description of the aforementioned security properties and methods to implement the same.

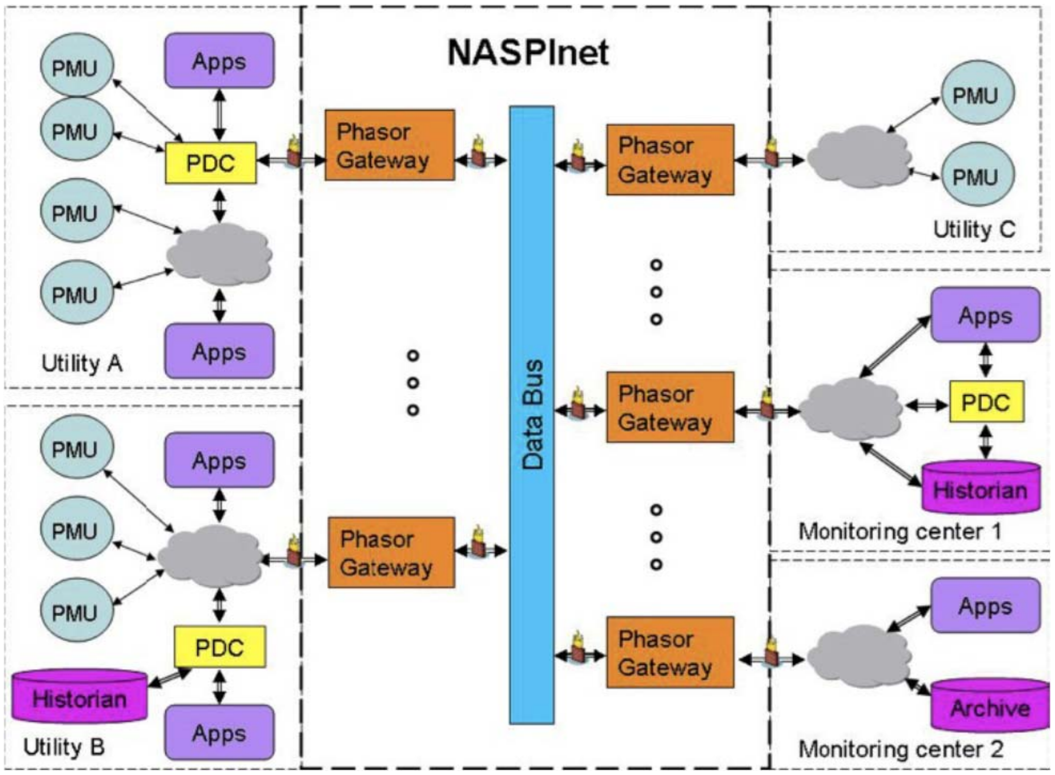

Figure 1: NASPInet conceptual architecture [2]

Confidentiality of data pertains to preventing it from being accessed by unauthorized entities. The PMU data contains sensitive information about the state of the grid which can be abused by malicious eavesdroppers to disrupt grid operation. These malicious agents may also modify the data, thus compromising its integrity and leading applications or operators into making catastrophic decisions. Hence, the PMU data must be protected end-to-end, i.e., from the sender PGW to the receiver PGW. The security of data, as defined above, depends on trust management, and key establishment and management. For example, only trusted entities should be allowed admission into the network over which data is transmitted. In order to meet this requirement, every data-requesting entity must be authenticated by verifying its digital credentials. Similarly, end-to-end confidentiality and integrity depend on pre-established cryptographic keys between the source PGW and the destination PGW.

Since the confidentiality and integrity of data depends on distribution of cryptographic keys for encrypting and decrypting data, the key management infrastructure should be fault tolerant. For a system to be fault tolerant, it should not have any single point of failure; i.e., it should be able to function even if any individual component undergoes failure. In regular key management protocols, the central entity responsible for distributing the keys, known as the key distributor (KD), is a point of failure. If the KD goes down, senders will not have access to new keys for encrypting data, forcing the sender to either use old keys or stop sending data, thus compromising either security or availability. One solution to this problem is replication of the KD, i.e., maintaining replicas to take its place in the event of its failure. Large Internet companies like Google and Facebook utilize large scale computing systems, known as web-scale architectures, to manage large volumes of data. These architectures are designed to be flexible, they provide near linear scaling, and they are leveraged to allow computations and optimizations. Hence, they not only provide reliability, but also deliver high performance. ZooKeeper is one of the services that enables these functionalities.

In this thesis, we propose the design for an interface that enables the group domain of interpretation (GDOI) key management protocol to replicate its key server by using the ZooKeeper coordination service. The thesis shall unfold as follows. In Chapter 2, we give a brief description of key management and the methodologies to perform the same. In Chapter 3, we discuss GDOI, a group-key management protocol, to be used for providing cryptographic keys to the PGWs. In Chapter 4, we elucidate the importance of replicating the GDOI KD, the group controller key server (GCKS). We then explain the important functionalities required for proper replication. In Chapter 5, we introduce ZooKeeper as a viable option to provide replication for the GCKS. In Chapter 6, we describe the design of the interface that allows running GDOI on ZooKeeper. In Chapter 7, we analyze the advantages of using ZooKeeper for replication over doing so ourselves. Finally, in Chapter 8, we present our conclusions and propose future work in the area.

2. KEY MANAGEMENT

Secure communications systems employ sophisticated protocols to protect the privacy of the data exchanged on it. These protocols involve the use of cryptographic primitives such as encryption and cryptographic message authentication codes. These primitives rely on cryptographic keys and they must be established between the sender and the receiver. The process of establishing these cryptographic keys between two entities and then maintaining them over their life cycle is called key management. For example, the sender may encrypt the data using a cryptographic key before transmitting it to the receiver. The keys that encrypt the network traffic (data) are called traffic encrypting keys (TEKs). Since the receiver must have the TEK to decrypt the message and retrieve the data, TEKs must also be transmitted on the network. Since malicious entities can easily access data without breaking the cryptographic algorithms if they gain access to the TEKs, it is of utmost importance to protect the TEKs as well. To prevent the TEKs from being compromised during transfer, they themselves are encrypted with key-encryption keys (KEKs). These KEKs in turn may be established using long-term keys that established using out-of-band mechanisms. Since the amount of traffic constituted by TEKs is much smaller than the amount of traffic constituted by the actual data, the KEKs do not have to be changed that often and are distributed less frequently than TEKs.

2.1 PKM (Pairwise Key Management)

Entities that use a symmetric cryptographic algorithm share a key to communicate securely. The sender can transmit the key to the receiver on a secure or out-of-band channel other than the channel used to transmit data. It could also split the key into multiple segments and send them over different channels. The receiver can then combine them to recover the original key.

Since every pair of users must exchange keys, the total number of key exchanges required in an *n*-entity network is $n(n - 1)/2$, or $O(n^2)$. This is manageable in small networks, but becomes a problem in large networks. In these cases, group key management (GKM), in which a trusted, central key distributor (KD) distributes keys over the network, may be more efficient. Since the KD has to maintain only one long-term secret key for each member, and each member has to maintain only one long-term SA with the KD, the overall number of keys is $O(n)$, thus making it more manageable. Since NASPInet must account for multiple PGWs subscribing to a single PGW, GKM is a much more efficient option than PKM.

2.2 GKM (Group Key Management)

In GKM, a trusted KD provides TEKs and the KEK (also known as the group key) to all members of the group. As mentioned in the previous section, it is a more efficient option than PKM for key management in the NASPInet framework. The fundamental security requirement of key management is ensuring that only the intended recipient gets access to data. In the case of group key management, the following procedures help in meeting this goal:

1. Member authentication: To prevent any intruder from impersonating a legitimate group member, each prospective member must be identified and authenticated to verify that it

actually is what it claims to be.

2. Access control: After authentication, the member’s join operation must be validated to control its access to group communication, especially the group key.

3. Rekeying: The TEK and KEK must be changed at regular intervals to safeguard their secrecy. Also, each key must be completely independent from any of the previous ones to prevent being formulated from them.

4. Perfect forward and backward secrecy: Only current group members must be able to access data. Hence, old members must be prevented from deciphering messages transmitted after their expulsion. This is called perfect forward secrecy, or leave secrecy. Similarly, new members must be prevented from deciphering messages transmitted prior to their membership. This is called perfect backward secrecy, or join secrecy. Rekeying in the event of a change in group membership is the principal mechanism employed to maintain perfect forward and backward secrecy.

2.3 System Model

We now describe a group key management model based on the IETF framework. Since a single source (sender) multicasts data to multiple receivers, we assume one-to-many multicast. The entities involved in the model are the KD, member senders and member receivers. Each member performs key and security association (SA) management functions with the KD for delivery and management of data encrypted by the group key. An SA is set of security parameters shared by network entities to support secure communication of information between them. It may include attributes such as cryptographic algorithm and mode, TEK, and parameters for the network data to be passed over the connection.

The member sender sends encrypted data packets to member receivers on the multicast group. The KD also sends control packets, messages containing commands, reminders, notifications, etc., to the members via a control channel. The control channel is also used for management of TEKs, KEKs, keying material and SAs. The model is shown in Fig. 2.

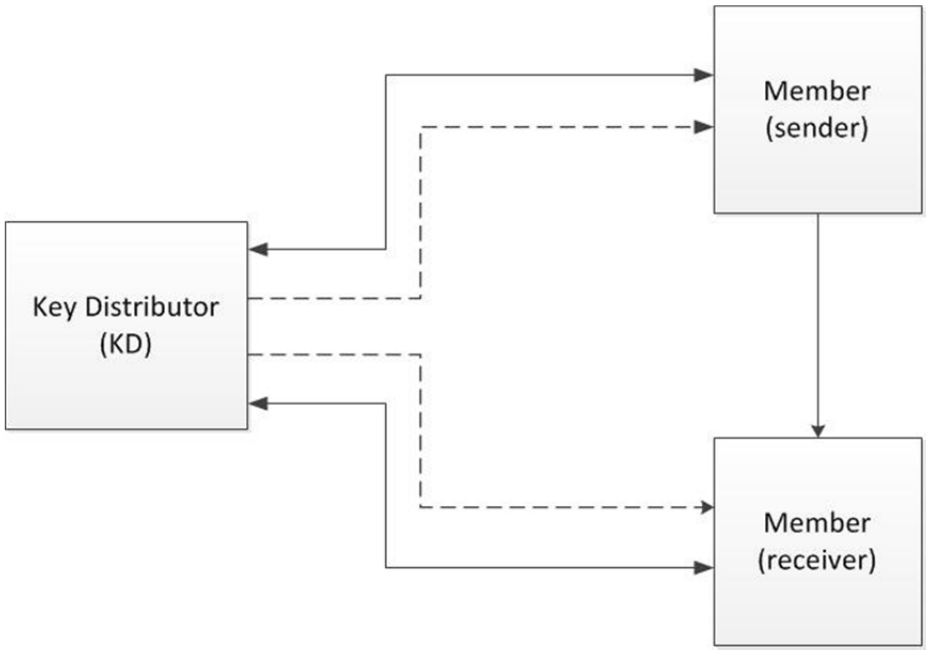

Figure 2: System model for GKM [3]

2.4 GSA (Group Security Association)

A GSA is defined to include an aggregate of three categories of SAs: Categories 1 and 2 between the KD and a member, and Category 3 among members.

1. Category 1 SA (or SA1) is required for the bidirectional unicast communication between the KD and a group member. It is initiated by a member to “pull” GSA information, including the SA, keys and the Category 3 SA, from the KD to either join the

group, or rejoin after getting disconnected. Hence, it is also referred to as pull SA or registration SA. This SA is known only by the KD and the corresponding member.

2. Category 2 SA (or SA2) is required for the unidirectional multicast transmission of key management/control messages from the KD to all group members. Since the control messages include the update or replacement of SA3, it can be said that SA2 is used to update SA3. SA2 is used by the KD to “push” rekeying messages and SA updates to the members. Hence, it is also known as push SA or rekey SA. This SA is known by the KD and all members.

3. Category 3 SA (or SA3) is required for the unidirectional multicast transmission from member sender to member receivers. Since it is used to secure the data traffic, it is also referred to as the data security SA. This SA is known by the KD and all members of the group. The processes taking place in a GSA are shown in Fig. 3.

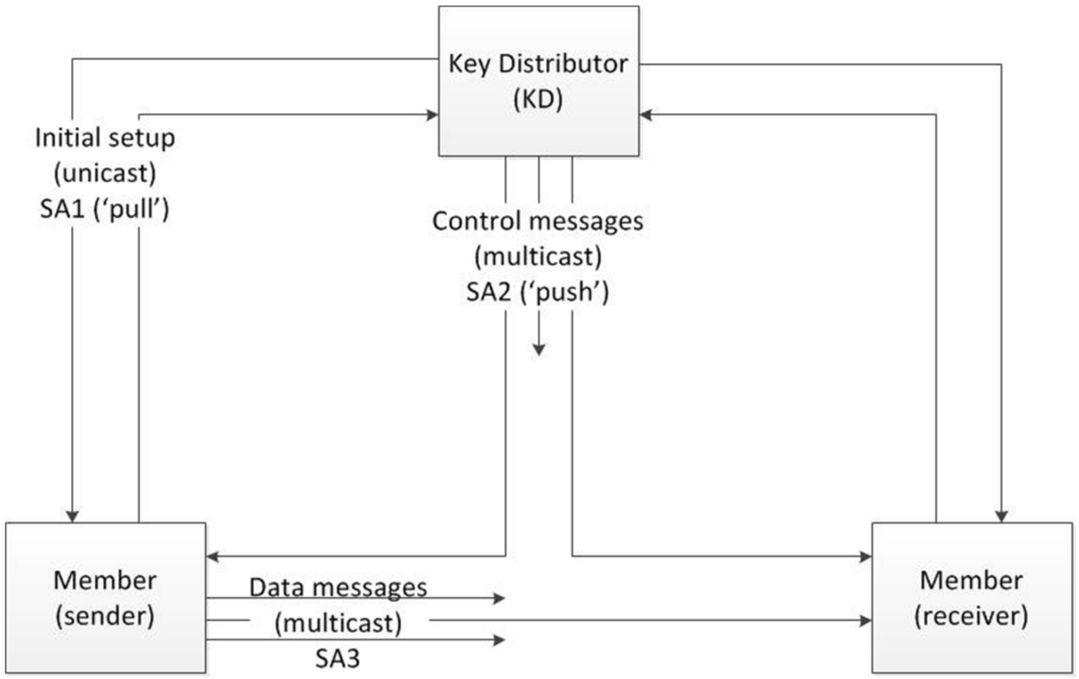

Figure 3: GSA definition [3]

3. GDOI (GROUP DOMAIN OF INTERPRETATION)

GDOI [4] is a GSA management protocol. It describes a set of procedures, message exchanges, and message payloads that govern the behavior of the entities in a group. GDOI messages are used to create, maintain, or delete security associations for a group.

GDOI uses ISAKMP's notion of domain of interpretation and applies it towards group communication. ISAKMP [5] is a key management protocol of the Internet Security Architecture (ISA) [6]. It defines the procedures for authenticating a communicating peer, creation and management of security associations, and threat mitigation. It thus provides a key management framework for transferring key and authentication data, independent of the key generation process.

GDOI is a two-phase protocol in which Phase 2 must be protected by Phase 1; i.e., Phase 2 exchanges may take place only after the Phase 1 exchange has completed. Phase 1, used to implement Category 1 SA, authenticates a client (member) to the KD, known as group controller/key server (GCKS) in GDOI parlance. GDOI uses the Phase 1 exchange from Internet key exchange (IKE) [7] for its own Phase 1 exchange. IKE is a widely deployed key exchange protocol, primarily used for IPsec (Internet Protocol Security). It builds upon the Oakley [8] protocol and ISAKMP, and is used to set up SA in the IPsec protocol suite. It is especially useful for protecting GDOI keying material as it is able to provide [9]:

i) peer authentication - via pre-shared keys or public key encryption

ii) confidentiality - via Diffie-Hellman exchange

iii) message integrity

a) man-in-the-middle attack protection

b) replay/reflection attack protection - nonce mechanism, hash-based message authentication code

c) denial of service protection - cookie mechanism

Phase 2 is newly defined in GDOI to cater to the GSA. It comprises two exchanges: GROUPKEY-PULL and GROUPKEY-PUSH. GROUPKEY-PULL, protected by the Phase 1 explained above, is responsible for creating Category 2 and Category 3 SAs. It enables the member to “pull” KEK and TEK keying material, policy, and attributes from the GCKS. The GROUPKEY-PUSH datagram, used to create or modify Category 2 and Category 3 SAs, is “pushed” from the GCKS to the members. The KEK protects the GROUPKEY-PUSH message. The various message exchanges that take place during the two phases of GDOI are shown in Fig. 4.

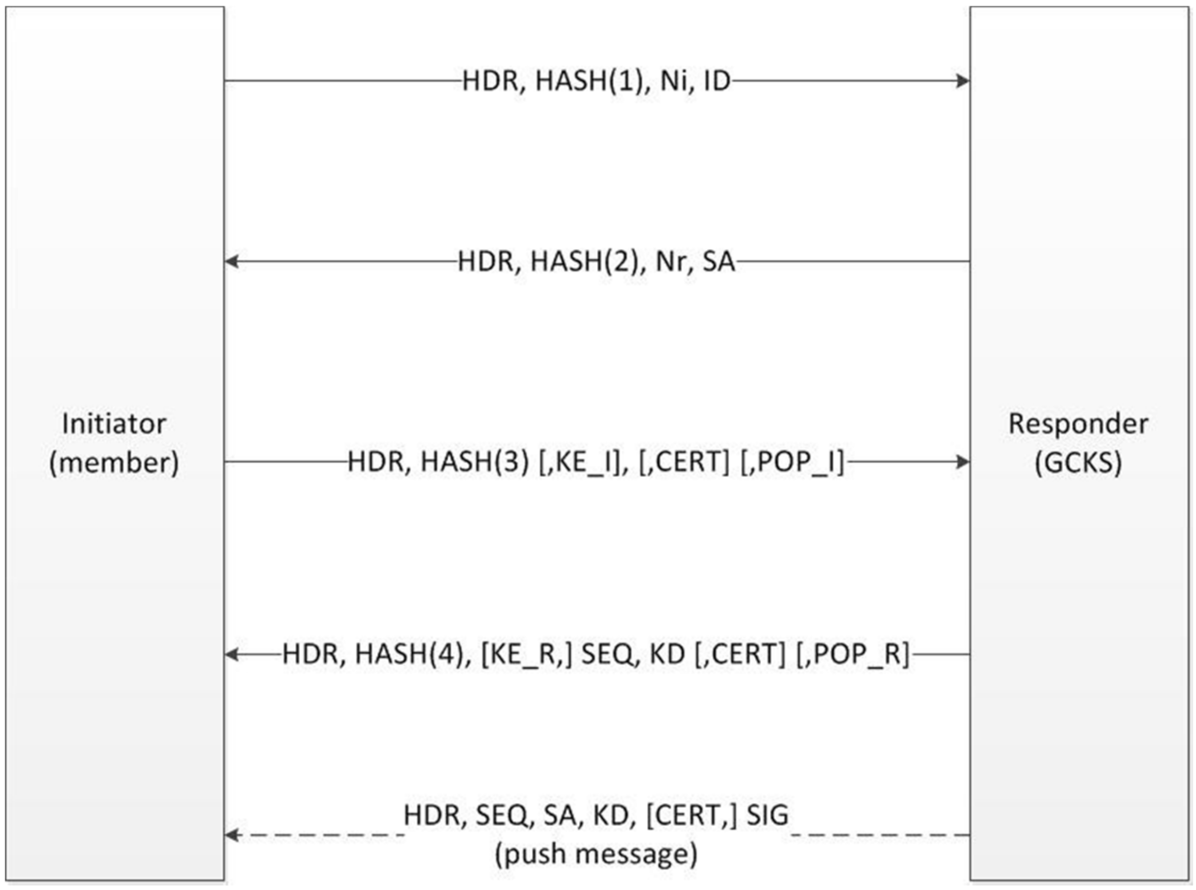
HDR, HASH(1), Ni, ID
HDR, HASH(2), Nr, SA
Initiator
(member)
HDR, HASH(3) [,KE_I], [,CERT] [,POP_I]
Responder
(GCKS)
HDR, HASH(4), [KE_R,] SEQ, KD [,CERT] [,POP_R]
HDR, SEQ, SA, KD, [CERT,] SIG
(push message)

Figure 4: GDOI Phase 2 exchange [3]

4. REPLICATION

High availability and confidentiality of data is one of the key requirements of the NASPInet infrastructure. To meet this requirement, the GKM infrastructure must be fault tolerant and ensure that network entities have access to TEKs for encrypting or decrypting data. To provide fault tolerance to the GKM infrastructure, we must eliminate its single point-of-failure, the KD. If the KD goes down, senders do not have access to new keys for encrypting data, thus compromising availability or confidentiality. Also, the receivers may not have access to the keys needed for decrypting messages, thus hampering availability. One way of preventing the KD's crash from affecting the system is to maintain its replicas, which can replace it as and when needed. We address replication of the KD in this work.

Replication, the process of storing data on multiple devices (usually a cluster of servers), is one of the most powerful tools to achieve high availability. The server providing a service may crash due to environmental hazards, operator errors, or infrastructure failures like power outages and software or hardware faults. In such a scenario, it is simply replaced by another server from the cluster. The successor is configured based on the state of its predecessor and the application is restarted on it. In addition to availability, replication also enhances system performance and scalability by exploiting locality of reference and read-intensive references; local servers store and deliver replicas rather than one global server for all the data. If not managed efficiently, replication can lead to issues like lack of consistency between the redundant sources, decreased system throughput, deadlocks, and security vulnerabilities. We shall now outline the main functionalities which must be provided to prevent consistency issues from arising due to replication. The main headings of the outline are: I. Atomic broadcast. II. Consensus. III. Leader Election. IV. Two-Phased Commit.

I. Atomic Broadcast [10]: Broadcast is the process of a server transmitting messages to the remaining servers in the cluster. A broadcast is atomic if it can guarantee that all servers in the cluster agree on the messages delivered, and on the order in which they are delivered. With this guarantee, consistency is trivially ensured if every operation on a replicated server is distributed to all replicas using atomic broadcast. We elucidate the conditions required for atomic broadcast, starting from basic broadcast [11].

1. Basic broadcast: A correct server would eventually deliver the message, where a correct server is one that has not crashed. Here, a sender delivering the message means the receiver receiving it.

2. Reliable broadcast: satisfies the following three properties:
i) Agreement: All correct servers deliver the same set of messages; i.e., if a correct server delivers a message m, then all correct servers eventually deliver m. Hence, a message is either delivered by all correct servers, or by none of them.
ii) Validity: This set includes all messages sent by the correct servers; i.e., if a correct server sends (broadcasts) a message m, the server will eventually deliver m.
iii) Integrity: Each message delivered from the set is identical to the one sent by its send-

er; i.e., for any message m, every correct server delivers m at most once and only if it was previously sent by its sender.

3. Ordered broadcast:

i) FIFO ordering: Messages are delivered in the order that they were sent; i.e., if a server sent message m1 before sending m2, then all correct servers deliver m1 before m2.

ii) Causal ordering: Each message is delivered only after all the messages it depends on have been delivered; i.e., if message m2 depends on message m1 (m1 causally precedes m2), then all correct servers deliver m2 only after they have delivered m1.

iii) Total ordering: All correct servers deliver all messages in the same order, and hence have the same “view” of the system; i.e., if a correct server delivers m1 before m2, then any other correct server that delivers m2 will deliver it only after delivering m1.

4. Atomic broadcast: A reliable totally-ordered broadcast is called an atomic broadcast.

II. Consensus

The consensus problem is for each server to propose a value in the form of a packet, a sequence of bytes, and for all of them to agree on one from those proposed. The agreement ensures consistency and is reached by voting. The minimum number of votes required for agreement is called a quorum. Usually quorums are required to have a majority number of servers. Quorums satisfying the aforementioned condition are called majority quorums. The unit of agreement is reached upon by exchanging packets with a quorum of servers is called a proposal. Common examples of consensus include totally ordered broadcast, in which the processes agree on the order of message delivery, and leader election, in which the servers agree on the leader.

III. Leader Election

In most clusters designed for high availability, the requests are processed by one master replica, known as the leader. The process of choosing the leader from among nominees is called election. The leader must be unique, and is therefore usually chosen as the one with the largest identifier (ID), which may be any useful value, as long as it is unique and totally ordered. The participants in the voting process must agree on the leader. They do so by deciding on the proposal with the highest ID. If at any point in time the current leader wishes to retire, then its replacement is found by the means of another election. The exact sequence of steps is as follows:

i) The leader (or nominee) sends each of its followers a NEW_LEADER proposal.

ii) The follower (or participant) acknowledges or ACKs the proposal.

iii) The new leader will COMMIT the proposal once a quorum of followers have ACKed it.

iv) The follower will commit any state it received from the leader when the NEW_LEADER proposal is COMMITED.

v) The new leader starts accepting new proposals only after the NEW_LEADER proposal has been COMMITED.

If an election terminates erroneously, the leader will not have quorum and hence the NEW_LEADER proposal would not be committed. At this point, the leader and its remaining followers will timeout and go back to election. It is also important to note that if a follower has proposals with IDs higher than that of the leader, they will be discarded. The rationale behind this is that since the leader, by definition, has the highest ID, the follower must have arrived after the election (or else it would have become the leader). Since the leader itself has not seen the proposal, it could not have sent it to the quorum. Hence, these proposals cannot be committed and are discarded. Therefore, the leader establishes an ID to start using for new proposals by setting its ID to be 1 more than the currently highest ID. This allows us to skip blocks of uncommitted proposals and not worry about duplicate proposals for a given ID.

IV. Two-Phased Commit

If a process crashes after sending its proposal to one set of servers, but before sending it to the other, the two sets of servers would not be able to decide the same value. The solution to this problem is atomic commit - the proposal is committed by either all servers or none. This ensures that none of the updates are partial.

A transaction comes to an end when a server requests it to be committed. A simple way to perform atomic commit is to keep communicating the commit request to all servers till they acknowledge that they have carried out the same. It is impossible to reach consensus in an asynchronous system if a node undergoes failure [12]. The two-phase commit protocol manages to reach consensus in asynchronous systems even in presence of a failure by masking it. It does so by replacing a crashed process with a new one whose state is set from information from other processes or saved in storage. In the first phase (voting phase), the leader asks the followers if they are ready to commit. Once it gets an affirmative response from a quorum of followers, it moves into the second phase (completion phase), telling them to commit.

The entire process comprises the following individual steps:

i) The leader sends proposals to its followers in the order that it receives requests.

ii) The followers receive proposals in the order that the leader sent them, process messages in the order that they receive proposals, and send an ACK to the leader in the order that they process messages.

iii) The leader receives ACKs from followers in the order that they sent it. Once it receives ACKS from a quorum, it issues a COMMIT to all followers.

iv) The followers process COMMITs and deliver the proposal message.

5. ZOOKEEPER

In the previous section, we described the functionalities required for ensuring consistency when performing replication. The complexity of coordination necessitates a service that can handle the coordination issues, while exposing an intuitive interface to the user. ZooKeeper [13] is one such service for coordinating processes of distributed applications with a user-friendly API. It is used by Yahoo! for its Fetching Service and Message Broker services, and by non-Yahoo! applications like Katta. Unlike Google's offering, Chubby, ZooKeeper does not make use of blocking primitives since they can cause slow or faulty clients to hamper the performance of faster ones. As a result, it is able to expose a wait-free and event-driven interface which enables the service implementation to have high-performance and fault tolerance. The performance aspects of ZooKeeper allow it to be used in large distributed systems, and the reliability aspects prevent it from becoming the single point of failure in big systems. Since it guarantees FIFO client ordering of all operations and linearizable writes, it meets the criteria to perform coordination.

ZooKeeper provides the user with primitives and guarantees, which we now outline under the following main headings: I. Sessions. II. Data Model. III. Guarantees.

I. Sessions

The ZK service comprises a set of servers, called ensemble, each of which must know about the rest. The service is available as long as the majority of the servers in the ensemble are available. A client connects to the ZooKeeper service and initiates a session.

Clients submit requests to ZK through the client API. The client library is responsible for exposing the ZK service interface through the client API as well as managing the network connections between the client and ZK servers. The client creates a handle to the ZooKeeper service using the list of servers in the ensemble. The ZooKeeper client library will pick an arbitrary server from the list and try to connect to it, transitioning the handle to CONNECTING state. If this connection fails, or if the client becomes disconnected from the server for any reason, the client will automatically try the next server in the list, until a connection is (re-)established, transitioning the handle from CONNECTING to CONNECTED state. During normal operation the handle is in one of these two states. Once connected to the service, the client library maintains a TCP connection between the client and the server through which the client sends requests, gets responses, gets watch events, and sends heartbeats. If an unrecoverable error occurs, such as session expiration or authentication failure, or if the application explicitly closes the handle, the handle moves to the CLOSED state.

II. Data Model

1. Znodes

ZooKeeper has a hierarchial name space, much like a file system, in which each node can have data associated with it as well as children. These nodes in the ZooKeeper data tree are referred to as znodes, as shown in Fig. 5.

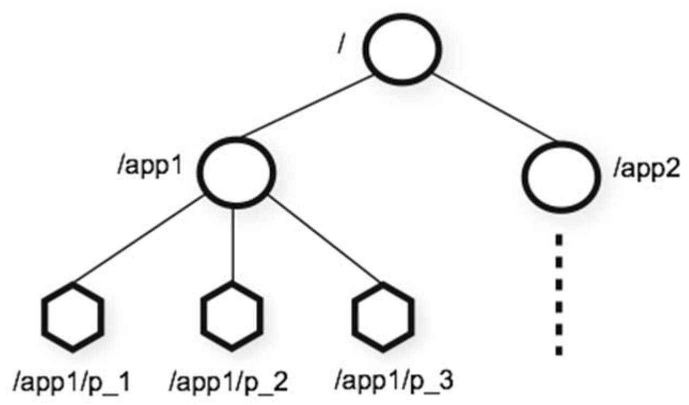

Figure 5: Zookeeper data model – znodes and namespace hierarchy [14]

The servers maintain an in-memory image of the data tree along with transaction logs and snapshots in a persistent store. Because the data is kept in memory, ZooKeeper is able to provide its clients with high throughput, low latency, highly available, strictly ordered access to the znodes. Paths to znodes are expressed as canonical, absolute, slash-separated paths. Znodes maintain a stat structure that includes time stamps as well as version numbers for data or acl changes. These allow ZooKeeper to validate the cache and to coordinate updates. Each time a znode's data changes, the version number increases. For instance, whenever a client retrieves data, it also receives the version of the data. And when a client performs an update or a delete, it must supply the version of the data of the znode it is changing. If the version it supplies does not match the actual version of the data, the update will fail.

i) Watches

Clients have the option of setting a watch on a znode while performing read operations. Watches are one time triggers; once a client gets a watch event and wants to get notified of future changes, it must set another watch. Because watches are one time triggers and there is latency between getting the event and sending a new request to get a watch, the znode may change multiple times between getting the event and setting the watch again. When a client disconnects from a server (for example, when the server fails), it will not get any watches until the connection is reestablished. For this reason session events are sent to all outstanding watch handlers, which can be used to go into a safe mode: The client does not receive events while disconnected, so its process should act conservatively in that mode.

With regard to watches, ZooKeeper maintains these guarantees:

a) Watches are ordered with respect to other events, other watches, and asynchronous replies. The ZooKeeper client libraries ensures that everything is dispatched in order.

b) A client will see a watch event for a znode it is watching before seeing the new data that corresponds to that znode.

c) The order of watch events from ZooKeeper corresponds to the order of the updates as seen by the ZooKeeper service.

ii) Data Access

The data stored at each znode in a namespace is read and written atomically - reads get all the data bytes associated with a znode and a write replaces all the data. Each node has an access control list (ACL) that restricts who can do what by specifying sets of IDs and permissions that are associated with those IDs. When a client connects to ZooKeeper and authenticates itself, ZooKeeper associates all the IDs that correspond to a client with the client’s connection. These ids are checked against the ACLs of znodes when a client tries to access a node. An ACL pertains only to a specific znode, not to its children.

Read requests sent by a ZooKeeper client are processed locally at the ZooKeeper server to which the client is connected. If the read request registers a watch on a znode, that watch is also tracked locally at the ZooKeeper server. Write requests are forwarded to other ZooKeeper servers and go through consensus before a response is generated. Sync requests are also forwarded to another server, but do not actually go through consensus. Thus, the throughput of read requests is directly proportional and that of write requests is inversely proportional to the number of servers.

iii) Ephemeral Nodes

ZooKeeper also has the notion of ephemeral nodes. These znodes exists as long as the session that created the znode is active. When the session ends the znode is deleted. Because of this behavior ephemeral znodes are not allowed to have children.

2. Time in ZooKeeper

ZooKeeper tracks time multiple ways:

i) Zxid - Every change to the ZooKeeper state receives a stamp in the form of a ZooKeeper Transaction Id (zxid). Each change will have a unique zxid and maintain total ordering; i.e., zxid1 < zxid2 => 1 happened before 2.

ii) Version numbers - Every change to a node increments its version number by 1. There is a version number associated with change in data (version), children (cversion), and

ACL (aversion).

iii) Ticks - When using multi-server ZooKeeper, servers use ticks to define timing of events such as status uploads, session timeouts, connection timeouts between peers, etc. The tick time is only indirectly exposed (half the minimum session timeout); if a client requests a session timeout less than the minimum session timeout, the server will tell the client that the session timeout is actually the minimum session timeout.

iv) Real time - ZooKeeper uses real time, or clock time, to put timestamps only during znode creation or modification.

III. Guarantees

1. Ordering guarantees:

i) Linearizable writes: Since ZooKeeper uses an atomic broadcast protocol, Zab [15], all requests that update the state of ZooKeeper are serializable and respect precedence. In addition to the guarantees provided by regular atomic broadcast, Zab also guarantees that changes broadcast by a leader are delivered in the order they were sent and all changes from the previous leaders are delivered to an established leader before it broadcasts its own changes. Also, since TCP is used for transport, message order is maintained by the network. The entire process from request to response is shown in Fig. 6.

ii) FIFO client order: All requests from a given client are executed in the order that they were sent by the client. This enables clients to submit operations asynchronously, and hence have multiple outstanding operations at a time. Hence, the linearizability in this scenario is called A-linearizability (asynchronous linearizability). This is different from the original definition of linearizability [16], in which a client may only have one outstanding operation at a time.

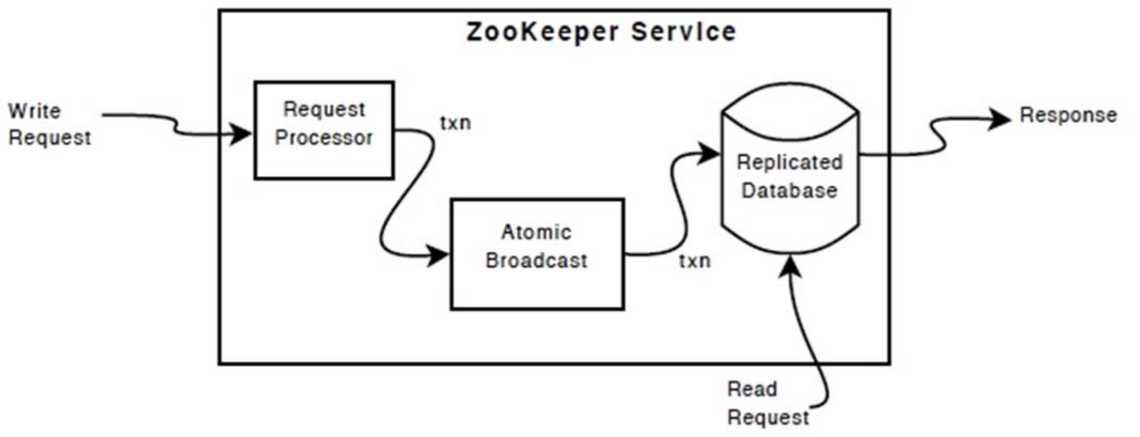

Figure 6: The write requests are sent to the leader which transmits it to its followers via atomic broadcast [14]

2. Consistency guarantees:

i) Sequential Consistency: Due to the total ordering guarantee, updates from a client will be applied in the order that they were sent.

ii) Atomicity: ZooKeeper messaging is very similar to two-phased commit. Hence, updates either succeed or fail - there are no partial results.

iii) Single System Image: A client will see the same view of the service regardless of the server that it connects to.

iv) Reliability: Once an update has been applied, it will persist from that time forward until a client overwrites the update. This guarantee has two corollaries:

a) If a client gets a successful return code, the update will have been applied. On some failures (communication errors, timeouts, etc.) the client will not know if the update has applied or not. ZooKeeper takes steps to minimize the failures, but the only guarantee is only present with successful return codes.

b) Any updates that are seen by the client, through a read request or successful update, will never be rolled back when recovering from server failures.
v) Timeliness: The clients view of the system is guaranteed to be up-to-date within a certain time bound (on the order of tens of seconds). Either system changes will be seen by a client within this bound, or the client will detect a service outage.

Due to the consistency guarantees, ZooKeeper can serve older data on receiving a read request. As a result, reads are faster than writes. Also, using these consistency guarantees, it is easy to build higher level functions such as leader election, barriers, queues, and read/write revocable locks solely at the ZooKeeper client.

6. DESIGN

We now describe how we can use ZooKeeper to provide replication services for GDOI. The GDOI controllers (operational and backup) and the GDOI members act as ZK clients, connecting to the ZK service, as shown in Fig. 7. Phase 1 of GDOI deals with authenticating a client before granting it membership. To prevent re-authentication, we save the SA information of this client to be used for verification in future. Phase 1 authentication involves trust establishment between the GKD (or whatever it is called here) and the group member, and the interaction in our approach is identical to the one in the original GDOI protocol.

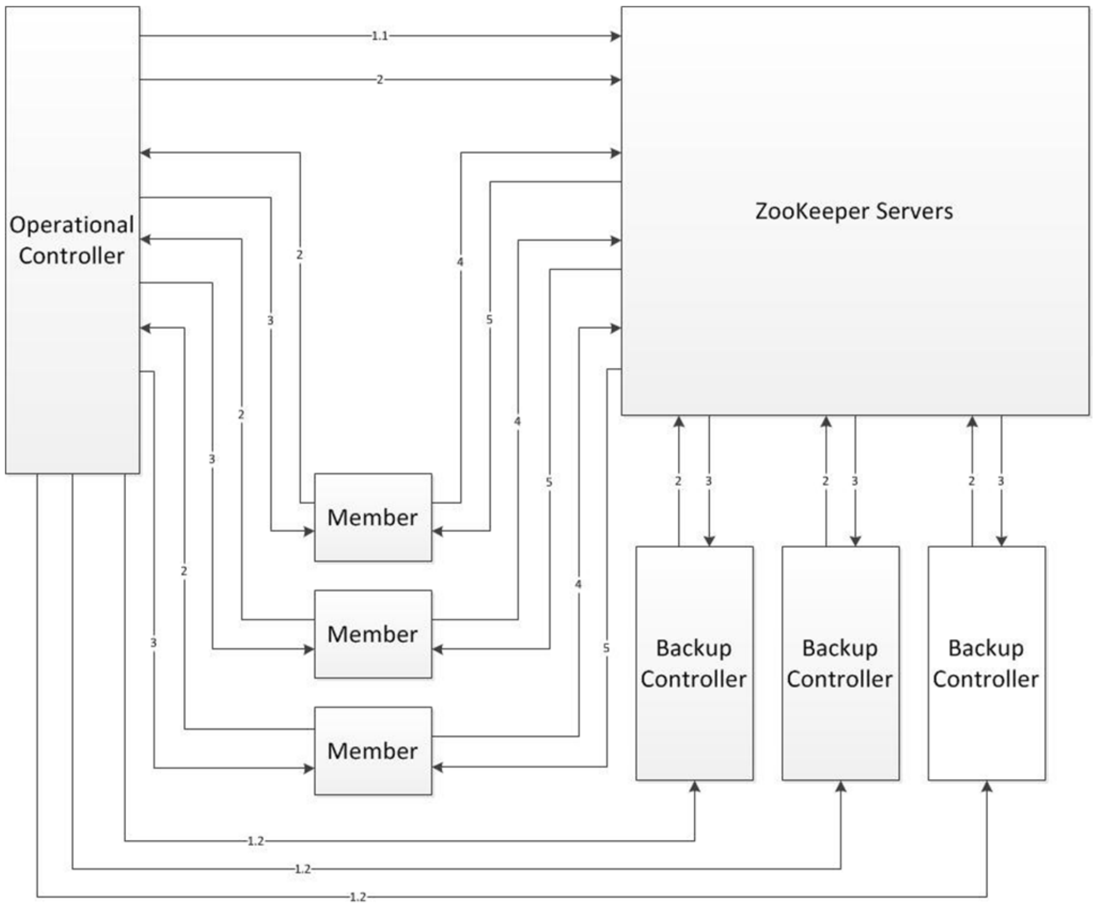

Figure 7: Architecture to use ZooKeeper for replicating GDOI GCKS

In Phase 2, the operational controller updates the keys by performing write operations, while the members receive these updates by performing read operations. In the event that the operational controller goes offline, one of the backup controllers becomes the new operational controller. Since the state of the system is maintained by the ZK service, the new operational controller can continue from where the previous one left off.

ZK service comprises a set of servers called the ensemble. All the servers in the ensemble must know about each other. They maintain an in-memory image of state, along with transaction logs and snapshots in a persistent store. The ZK service will be available as long as a majority of the servers are available.

We begin with the first phase of ZK messaging, leader activation. In this phase, ZK elects a leader from the ensemble and waits for followers to connect to the leader. The ensemble is now ready for the second phase of ZK messaging, active messaging. In this phase, the elected leader accepts messages to propose and coordinates their delivery to its followers, as shown in Fig. 8.

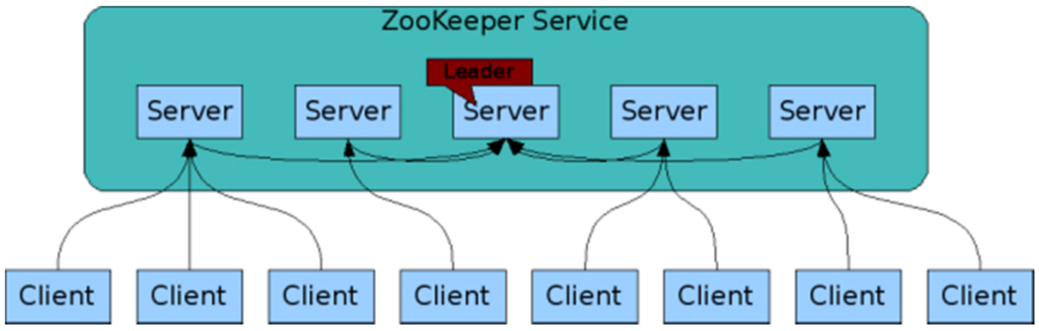

Figure 8: All requests received by the service are forwarded to the leader, which then sends out message proposals to its followers for agreement [14]

The operational controller now connects to the ZK service. A ZK client must have a connection string containing a comma-separated list of host/port pairs, each corresponding to a server in the ZK ensemble. Once connected to the ZK service, the operational controller creates a regular znode and an ephemeral znode in the data tree. We refer to the regular znode as the TEK znode and the ephemeral znode as the expiry znode. Our motivation for making the expiry znode ephemeral will become clear in the following paragraph. The operational controller now sends the connection string and the path names of the expiry znode to each backup controller.

Using the connection string sent to them by the operational controller, the backup controllers connect to the service and set a watch on the ephemeral znode created by the operational controller. In the event that the operational controller suffers a crash and fails to recover within the session timeout, the session is said to have expired. On session expiration, all ephemeral znodes associated with the session are deleted and all clients watching the znode are notified of the change. Hence, if the operational controller fails to recover within session timeout, backup controllers are notified of the same, and the next available one in the predetermined succession order takes over.

Now, the operational controller listens for incoming requests from prospective members, authenticated by Phase I. Upon receiving a request, it launches a separate thread for the particular member.

In each thread, the controller reads data from the respective member to check whether the request made is to join or leave the group. In the event of a join request, it creates a znode in the ZK object data tree with a unique path name. This is the member's KEK znode. The controller adds the member's authentication credentials to the ACL of its KEK and TEK znodes. It then sends the connection string and the path names of the TEK and KEK znodes to the member and starts updating the KEK and TEK periodically. ZK ensures that once an update has been applied, it will persist until a client overwrites it. This means that if the controller gets a successful return code, the update has been applied. Also, since the broadcast is atomic, all of the data is written and hence local replicas in the server database never diverge.

TEK updates are frequent, but require performing a write operation on only one znode. KEK updates require performing a write operation on every znode of the group, but do not occur frequently. Hence, TEK updates are frequent but involve performing a computationally low-cost operation, and KEK updates involve performing a computationally high-cost operation, but occur occasionally. Therefore, the computational cost of the process is relatively low for the most part with occasional spikes.

The member, now connected to the controller and receiving information from it, connects to the ZK service using the connection string. Since the member's credentials match those in the ACL of its KEK and the TEK znodes, it can read data from the particular znode. The member sets a watch on the TEK and KEK znodes. Every time the controller writes an update to either of them, the watch set on it is triggered, sending a notification to the member. The member reads the update, resetting the watch for the next update, and makes a note of it in its log. Since updates from a client are applied in the order that they were sent, the members receive keys in the order that the controller sent them. Also, since the read and write operations are atomic, i.e., write replaces all the data in a znode and read receives all the data associated with it, the results are always complete, never partial. Furthermore, since ZK guarantees a single system image—i.e., a client will see the same view of the system irrespective of the server it connects to—it is going to be the same key. Also, if a member gets disconnected, upon reconnection it will view the same state as earlier.

7. ANALYSIS

In this section we elucidate the advantages of running GDOI with ZooKeeper as a replication service over running GDOI on its own.

1. Increased availability

As mentioned earlier, GDOI has a single point of failure: the GCKS. Since GCKS is the only entity that is responsible for key distribution, if it crashes, members are unable to receive any keys until it recovers from the crash and starts to function again. In the replicated model, the backup controllers can take the place of the operational controller in the event of its failure. The design provides for the replacement by a backup controller to be quicker than the recovery of the operational controller.

2. Eliminating need for re-authentication

In GDOI, when a GCKS resumes operations after recovering from a crash, it must establish new TEKs and KEK. This off-schedule update makes the member realize that the GCKS had crashed. On the other hand, in our proposed design, since key updates are time stamped, the GCKS does not have to send the TEKs and KEK until they are due for the update. This not only saves GCKS from the computationally intensive process of brute force key-update upon recovery, but also makes GCKS failure almost invisible to the members. Also, since ZooKeeper handles coordination and provides consistency guarantees, each backup controller is ensured to have the same view of group membership.

3. Higher groupkey-push throughput

In the GDOI scheme, a single GCKS handles the entire load of providing group members with their respective key updates. In the replicated system, the GCKS writes the key updates to the ZooKeeper service, which replicates it across the servers in its ensemble. Since the load is now spread over a larger number of servers, the throughput of the groupkey-push operation is higher; i.e., more members can receive the groupkey within the required time bound. ZooKeeper also performs load balancing, i.e., distributing the load evenly among servers, thus maximizing throughput.

4. Ease of implementation

Replication can lead to problems like inconsistency between redundant sources, deadlocks, decreased throughput of operations, etc. To prevent such issues from arising, providing coordination services is of utmost importance. ZooKeeper provides common coordination services such as synchronization, consensus, leader election, etc., thus eliminating the need for implementing these from scratch. It exposes a simple API which enables developers to easily implement primitives based on their needs. The client library handles network connections, thus absolving the client of managing the same.

8. CONCLUSION AND FUTURE WORK

In this thesis, we focused on security and availability of data being exchanged between the PGWs, as a part of the NASPInet. We highlighted the importance of key management for maintaining confidentiality and integrity. We discussed the choice of group key management over pairwise key management, and discussed GDOI as the suitable protocol for implementing the same. We described the need for replication to ensure high availability, and the conditions that must be met for the replicated system to be dependable. We proposed the use of ZooKeeper for replication, explained its internal workings and the primitives provided by its API, and how it can be used to provide a dependable replicated system. We finally proposed a design for the interface that can allow GDOI to use ZooKeeper as a replication service.

For future work, we would like to verify that running GDOI on ZooKeeper can not only provide high availability, but also meet the performance benchmarks as specified by NASPInet. We intend to do so by running it on a large-scale network simulator. Specifically, we would like to quantify and minimize the following parameters:
i) the percentage of updates read incorrectly by the member
ii) the time elapsed between the controller writing an update to the ZooKeeper service and the member reading that update
iii) the time elapsed between the operational controller going down and a backup controller taking its place

REFERENCES

[1] North American SynchroPhasor Initiative. (2008, May). Specification for North American SynchroPhasor Initiative Network. [Online]. Available: https://www.naspi.org/site/Module/Team/dnmtt/naspinet/quanta_sow.pdf

[2] R. Bobba, E. Heine, H. Khurana, and T. Yardley, "Exploring a tiered architecture for naspinet," in *Innovative Smart Grid Technologies (ISGT), 2010*, Jan. 2010, pp. 1–8.

[3] T. Hardjono and L.R. Dondeti, *Multicast and Group Security*. Norwood, MA: Artech House, 2003.

[4] M. Baugher, B. Weis, T. Hardjono, and H. Harney. (2003, July). The group domain of interpretation (GDOI), RFC 3547. [Online]. Available: http://tools.ietf.org/rfc/rfc3547

[5] D. Maughan, M. Schertler, M. Schneider, and J. Turner. (1998, Nov.). Internet security association and key management protocol (ISAKMP). RFC 2408. [Online]. Available: http://tools.ietf.org/html/rfc2408

[6] S. Kent and K. Seo. (2005, Dec.). Security architecture for the internet protocol (IP-Sec). RFC 4301. [Online]. Available: http://tools.ietf.org/html/rfc4301

[7] D. Harkins and D. Carrel. (1998, Nov.). The internet key exchange (IKE). RFC 2409. [Online]. Available: http://tools.ietf.org/html/rfc2409

[8] H. Orman. (1998, Nov.). The oakley key determination protocol. RFC. [Online]. Available: http://tools.ietf.org/html/rfc2412

[9] C. Kaufman, Ed. (2005, Dec.). The internet key exchange (IKEv2) protocol. RFC 4309. [Online]. Available: http://tools.ietf.org/html/rfc4309

[10] L. Rodrigues and M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1206–1217, Sept.-Oct. 2003.

[11] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Prentice Hall PTR, 2001.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, April 1985. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free coordination for internet-scale systems,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’10, 2010, pp. 11–11. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855840.1855851

[14] Apache ZooKeeper Overview. (2011, Mar.). [Online]. Available: http://zookeeper.apache.org/doc/r3.3.3/zookeeperOver.html

[15] F. Junqueira, B. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *41st International Conference on Dependable Systems Networks (DSN), 2011 IEEE/IFIP*, June 2011, pp. 245–256.

[16] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990. [Online]. Available: http://doi.acm.org/10.1145/-78969.78972