XanaduRFT (Community) User Guide

Version 1.0

Xanadu Big Data, LLC

Contents

I. Introduction

Xanadu is a next generation storage platform technology built for providing a highly scalable, available, consistent and fault tolerant NoSQL data store in a world of Big Data and extreme data resilience. Xanadu provides a unique combination of following features.

1. Unlimited Resilient Data Storage Capacity

Xanadu is a distributed data storage and query system that horizontally scalable to thousands of storage nodes. The storage nodes communicate each other via parallel network connections. Xanadu has no central or single master node, and thus, no single point of failure. Data is made available in the face of failures by using replication. Unlike most other NoSQL data storage systems, the number of redundant copies is configurable by clients at storage time rather than fixed by system architects when the store is created[1].

Because the Xanadu data store uses an addressing scheme based on the data content, multiple clients can read or write data from the correct distributed nodes without needing to refer to any central index. Xanadu's patented [2] data placement strategy among distributed storage nodes allows architects to minimize operational concerns such as cascading failures[3] and correlated failures[4]. The Xanadu data store also runs a constant data "healing" process that ensures that even if a storage node fails the required number of redundant copies of all data that node was responsible for is recovered by making new redundant copies on the remaining nodes. As a result, Xanadu provides consistent view of the data in the face of multiple node and network failures.

In terms of the famous CAP theorem[5] Xanadu is "CP" (Consistent and partition tolerant). The number of failures Xanadu can tolerate before becoming unavailable for reads and/or writes is configurable. Xanadu remains available for data reads even with extreme and widespread

[1] The system architect can specify a default replication count for when the clients don't have a preference.

[2] US10067719: Methods and systems for storing and accessing data in a distributed data storage system

[3] The failure of one node in a distributed system means the clients all move on to the next node, which then can fail under the extra load - and so on taking the whole system down.

[4] If two or more stored copies are on nodes that tend to fail together, rather than independently, the data is more likely to become unavailable.

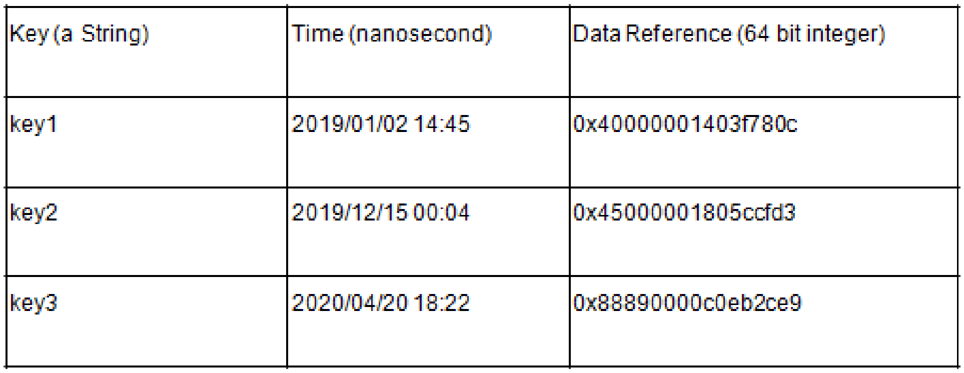
[5] https://en.wikipedia.org/wiki/CAP_theorem

failures – a feature made possible by Xanadu's new patented [6] data model – *timeline* data model.

2. Timeline Data Model

Xanadu takes the most popular NoSQL data model, the *key-value* store, and extends it with the concept of a *timeline* store - arbitrary keys (i.e. a word or *string*) are mapped to a timeline of values and a timestamp of when they were applied. In contrast to conventional key-value stores, timelines make a complete history of each key available to be queried and recalled at any time.

In terms of a traditional relational data model schema, the Xanadu's *timeline* data model splits the traditional key-value store into two main tables that represent a normalised version of the traditional relational data model schema – Key-Reference Store and Data Store.



Key (a String)	Time (nanosecond)	Data Reference (64 bit integer)
key1	2019/01/02 14:45	0x40000001403f780c
key2	2019/12/15 00:04	0x45000001805ccfd3
key3	2020/04/20 18:22	0x88890000c0eb2ce9

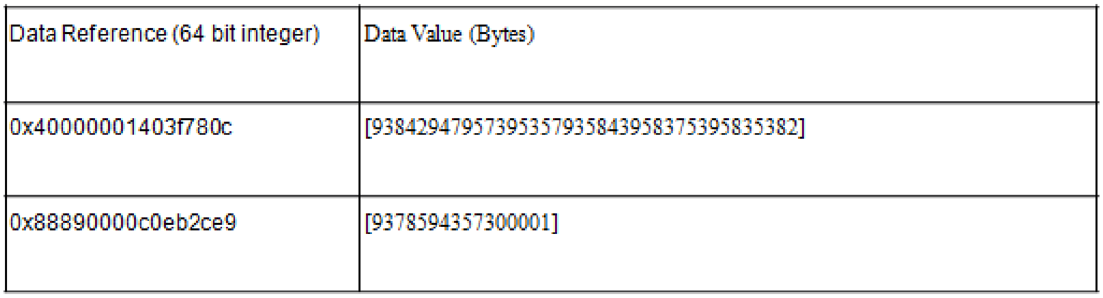
Key-Time-Reference Table

The upper table is called the Key-Reference Store and contains the keys and a reference to their value at a given time (the time the reference was assigned to the key). Data is only ever appended to this table, so the full history of a key's value(s) is contained in this log. Consequently a large fraction of this log can be stored in RAM on the node where it supports fast queries for clients. Entries are also guaranteed to have a unique time, accurate to the nanosecond, so that it is always possible to be sure in which order assignments were made.

[6] US10xxxxxx: Methods and systems for resilient, durable, scalable, and consistent distributed timeline data store

Xanadu selects unique timestamps for all atomic data updates (i.e. transactions), using a combination of the local system clock and counter. The current time (in milliseconds) is shifted left by 20 bits and the lower 20 bits reserved for a local counter that increments by 1 for each time value that is required within the same millisecond. This combination of real and logical time implies Xanadu has capacity for up to 1 million updates each millisecond. With each update getting a unique time, the state of the data at any point in time is uniquely specified and unambiguous.

The data reference is a unique numeric index that indicates an address or a pointer to the data value the second table, called the Data Store as shown below.



Data Reference (64 bit integer)	Data Value (Bytes)
0x40000001403f780c	[93842947957395357935843958375395835382]
0x88890000c0eb2ce9	[9378594357300001]

Reference-Value Table

Data stored in the value column in the Data Store is given a unique reference number. If a second block of identical data is stored it will be assigned the same unique reference and not stored a second time. Data in the Xanadu data store is inherently de-duplicated, and references once set are never deleted.

A data reference is a 64-bit number created using a combination of information to ensure effective deduplication (for identical storage requests in the future) as well as global uniqueness. The parts of this single number are:

1. Part of the full MD5 hashcode of the data (typically 32 bits).
2. The issuing node’s unique ID (as issued by the registry), typically 16 bits.
3. A sequence number, stored and incremented locally, to enable disambiguation of data blocks with identical hashcodes (typically 13 bits).
4. A replication number (typically 3 bits) that indicates how many redundant copies of this block there should be at any one time.

The immortality of the data makes it safe to cache data (and associated references) at any point in the system, and for any length of time. There is no need to deal with data expiring in the cache, so a number of efficient strategies are possible that enable Xanadu to provide an extremely fast data processing model.

The Key-Reference store supports *long running, non-blocking* atomic updates to multiple keys – something unique amongst even the most advanced NoSQL products. For conventional data models that represent only a single (i.e. current) state of the data, a transaction with many data edits causes latency issues while it is loaded into the database over the network. During this time other reads and writes must wait (i.e. be "blocked") until the large transaction has been uploaded and applied.

Xanadu's timeline model permits long-running transactions without blocking by using a condition that specifies exactly what part of the data model (i.e. a time and key range) must not be updated for the transaction to succeed. Intermingled writes either occur outside the conditional range, in which case the transaction can still succeed once uploaded, or, if inside the range the transaction can be failed immediately. Long-running transactions in Xanadu therefore succeed or fail because of designed model constraints rather than arbitrary time or size limitations.

When compared to other distributed database technologies, the timeline model enables Xanadu to provide superior support for high-volume concurrent queries and updates, long running or otherwise. Because the timeline model reflects reality so closely it is easy for application developers to reason about the system and build software that acts correctly even in rapidly changing and/or conflicting situations. Xanadu consequently supports complex use-cases like no other database.

3. Globally Atomic, Consistent, Isolated and Durable (ACID)

Xanadu provides non-blocking, long-running *Atomic, Consistent, Isolated and Durable (ACID)* multiple key updates for globally atomic, consistent transactions coupled to a simple query interface in detail:

- **Atomicity**–multiple data edits (updates or writes) will either fail or succeed together – no client will see a "part complete/part applied" data state at any time.

- **Consistency** - all clients will always receive the same results when querying the data model. No client will ever see some data, and another not see it (or equally see a different or outdated value).
- **Isolation**–concurrent transactions have the same combined effect on the data as if they were all executed serially, one after another. This is especially useful for reasoning about conditional updates (change value to X if K=5), as the value of the condition (k=5) might be being changed during the execution of this transaction - Isolation means it will be always clear what the value of K was at the time it was examined.
- **Durability** - data, once committed, will never change (even if there is a storage failure) except by proper application of the system API i.e. values don't change randomly or by some alternative external means.

The Key-Reference store represents the distributed, resilient, consistent (i.e. ACID) log of all the edits applied to the data. As the timeline is fixed once written, it too can be stored resiliently to disk and only the most frequently used information retained in RAM. Fast access to the most recent timeline values is the common case, and with the working copy loaded in RAM, Xanadu's distributed agreement algorithm can operate at maximum speed without needing to wait for slow disk IO operations. This makes Xanadu's read/write speed one of the fastest of all available systems.

II. Implementation

Xanadu implementation is currently split into three parts:

1. **XanaduRFT**: The Xanadu implementation that uses the RAFT[7] distributed consensus protocol for resilient, consistent Key-Value updates in the face of node failure.
2. **XanaduPX**: An updated version, based on XanaduRFT, which uses the PAXOS[8] distributed consensus protocol instead for Key-Value updates.
3. **XanaduNext**: A new implementation of the Xanadu codebase that is refactored to take the best aspects from the above two versions and anticipate the full range of capabilities planned and patented [9] for the platform.

[7] https://en.wikipedia.org/wiki/Raft_(computer_science)

[8] https://en.wikipedia.org/wiki/Paxos_(computer_science)

4. **XanaduRFT (Community)**: XanaduRFT (Community) is essentially the same as XanaduRFT except it has some limitations to encourage users to use the full XanaduRFT or XanaduPX when moving from prototype applications into production.

All versions have the same fundamental components, each made up as a single process suitable for deployment on different physical servers to minimize the impact of any single machine failure[10]. These components are:

1. **Registry Nodes:** these hold critical system information that all other nodes use to operate correctly. The registry nodes operate independently and identically, and the whole system can continue operating as normal if even only one registry is available.
2. **Key-Reference Nodes:** as a group these implement the distributed agreement mechanism that allows Xanadu to update the current data model with new values (i.e. key-reference edits), even if one or more participating nodes fails randomly. *Resilient* in this context means the updates will not be lost or corrupted if one or more of these nodes fail or a network partition occurs – this property is essential guarantee consistency[11]. Using either RAFT or PAXOS algorithms for this there needs to be an odd number of key-reference nodes, with a minimum of 3 to provide resilience to one node failure or network partition.
3. **Store Nodes:** these actually store the data on their local disk. Operating autonomously, they use the registries to discover each other and then handle the deduplicating distributed storage tasks (including auto healing).

1. Xanadu Registry

The registry is made up of one or mode nodes, as configured at design time by system architects by creating a config.js file that each registry node has a copy of. An example config file is shown below:

```
created= "2020/04/20";
```

[9] US10158483: Systems and methods for efficiently and securely storing data in a distributed data storage system; US10275400: Systems and methods for forming a fault-tolerant federated distributed database

[10] For testing purposes, each of Xanadu's components can be run as separate processes on a single machine.

[11] If a failure loses an update, even temporarily, clients might receive inconsistent values from other nodes in the system until the faulty node or network is repaired. Resilient agreement groups prevent this and ensure consistent results to clients even during a partial system outage.

```
xanaduID= "7AB7672888BCF";
distributedRegistryAddresses = "202.168.2.4:2020 202.168.4.5:2021
202.189.2.100:3002:2022";
```

This file identifies the Xanadu instance with the fundamental system constants – the "created" time of and the "xanaduID" (a 64-bit long integer random number in hexadecimal). These two numbers should be globally unique and immutable once the Xanadu instance has been created - all clients and other servers/nodes refer to these values to ensure they connect to servers/nodes that are part of their intended instance.

The distributedRegistryAddresses variable shows where the multiple registries are on the network as a space-separated list of IP addresses and port combinations. These are selected at design time and cannot change during operation, although they can be altered by shutting all Xanadu nodes, adjusting the required IP addresses on the network and then restarting *all the* nodes with new IP addresses in this edited config file.

The registries keep track of the storage nodes available in the system. When a storage node is started for the first time it is given a list of the Xanadu registries' IP addresses which it cycles through to register as a new node. The first registry that receives this new store's registration request issues it a unique ID (a random 64-bit integer) and, if not chosen by the system architect, a *store index*; a simple integer starting at zero. Alternatively, a node ID can be chosen by the system architect with help from the patented[12] *node placement* algorithm that assigns the next best node ID with a view to minimizing the chance of correlated node failures.

The registries maintain a list of all storage nodes that have contacted them and, because the store nodes have the (fixed) list of registries, they can periodically contact all registry nodes to ensure their presence is recorded. If a storage node becomes unavailable it will not contact the registries and it's "liveness" (as measured by the last contact time) will gradually move it down each registry node' priority list. With multiple storage and registry nodes, the former all in constant contact with each of the latter, the whole system can remain effective in the face of multiple server failures and network interruptions (often referred to as *partitions*).

[12] US10067719: Methods and systems for storing and accessing data in a distributed data storage system

Clients that wish to store data first contact one of more of the registries to retrieve the fixed configuration data as well as the current list of storage nodes. Clients use the node placement algorithm to derive a list of node indices that can have that particular data block. This derivation depends on the data block content, so two clients wanting to store the same data will derive the same list of node indices and therefore contact the same storage nodes. Storage nodes always check whether they already have a particular data block before storing it and return the previously allocated *data reference* for any data it already has.

Clients can derive the list of nodes independently and without reference to any external “master” index, and they can contact all storage nodes independently, therefore all data reading and writing can proceed without any network “bottlenecks”.

The registry creates a local directory called *XanaduRegistry* in which it creates a simple file Xanadu store. This simple version stores timelines of keys in a durable but not redundant or fault tolerant way. The key-reference log of this store is stored in the ***.kvs** files, and the data referred to therein is stored in the subdirectories (labelled with the unique Xanadu store ID and registry node number) as ***.sfs** files. The registry mainly uses this store to save the periodically updated registration state of the storage nodes throughout the system.

2. Key-Reference Nodes

These nodes collectively implement the distributed consensus algorithm necessary for fault-tolerant (i.e. resilient) updates of any key’s reference value.

Distributed consensus uses majority voting to get a clear result even if one (or even more) nodes fail or become uncontactable because of a network partition. Consequently, there needs to be an odd number of nodes in the group so that there is always a clear majority and never a tie. The minimum number is 3 to allow for 1 node failure, or a network partition that leaves 2 of the 3 nodes able to communicate (temporarily or permanently isolating the third). XanaduRFT and XanaduPX use different algorithms to achieve the consensus, each is detailed in the following sections.

XanaduRFT

The XanaduRFT uses the RAFT[13] algorithm and a randomly distributed arrangement of RAFT groups for different hash-based ranges of the key space, similar to the MultiRaft[14] implementations of CockroachDB and TiDB. For historical reasons these Key-Reference nodes are also known as the Xanadu *realtime nodes* and they read their configuration from a separate text file called XanaduRealtime.cfg. An example of this file is shown below:

```
groupSize=3
storageReplicas=2
node=202.168.2.4:3000
node=202.168.4.5:3001
node=202.189.2.100:3002
```

RAFT achieves fault tolerant agreement on updates by using “majority voting”, so there must be an odd number (3 or greater) of *nodes* listed in the file. The *groupSize* specifies how many nodes are in the overall group (3, 5, 7, etc) while the *storageReplicas* specifies how many copies of the accumulated update log should be saved into the underlying Xanadu data store. To prevent inconsistent results about the past history of all keys and implement a consistent *Timeline* store, the realtime nodes retain all agreed updates in local RAM memory, but periodically write them to the Xanadu storage nodes in blocks to prevent RAM filling up[15]. The *storagereplicas* determines how many redundant copies of each of these blocks should be stored – more copies imply higher resilience to node failures but a consequentially reduced update speed.

The RAFT algorithm works with a strong *leader* – a single node elected to take the primary contact from clients and handle most of the work, with the other nodes acting as *followers*. If there is only one RAFT group then the leader node can become overloaded with work, while the followers remain relatively idle. High-performance systems implement MultiRaft, where each node runs a number of *virtual-nodes* that form different RAFT groups for different ranges of keys. Each of these groups will have their own leader and so, on average, each of

[13] https://raft.github.io/

[14] http://sergeiturukin.com/2017/06/09/multiraft.html

[15] The realtime nodes can hold a very large number of updates in RAM due to the efficient representation of an update/edit. This facilitates fast queries as most recent information will be held in RAM on the realtime nodes.

the main (physical) nodes will, for a RAFT implementation a group size of 3, act as leader for about 1/3 of the key ranges and follower for the other 2/3. This balances out the read/write load amongst the RAFT nodes.

Therefore, XanaduRFT is capable of very high read/write rates using this approach. Agreed details are stored in memory (RAM) as a continuous *log* on the separate RAFT nodes until the log size exceeds a threshold. Periodically, XanaduRFT nodes store chunks of their edit log on the storage nodes and report the *data-reference* for these blocks of data to the Xanadu Registry nodes for resilient storage (another reason there should be more than one registry). Once the storage nodes acknowledge the store of a log block on multiple systems (typically 3 independent copies), and more than 1 registry has acknowledged the storage of the resulting data-reference for this block, the RAFT node can remove the block of data from RAM to make space for more updates.

As these updates are small 32 byte values the RAM of the RAFT node can accommodate a very high read/write rate while all these background storage operations are completed. Unfortunately, because of the distribution of key ranges into pseudo-random hash ranges (as per the MultiRaft methodology), it becomes very difficult to achieve multi-key atomic updates i.e. *transactions*. This is because a transaction typically involves keys in many different key ranges, and thus, requires complex update coordination across multiple RAFT groups in a MultiRaft system.

Because the key-reference log is stored in the data store, and the log references in the registries, query performance can suffer while the client retrieves these details when running a query over the dataset. To speed this process, and save unnecessary network traffic, XanaduRFT client driver code creates a local cache of the keys (i.e. strings) and the key-reference log. In the *XanaduCache* directory, the former appears as ***.xkc** and the latter as ***.xkle** files.

XanaduPX

Instead of overlaying complex protocols on top of MultiRaft to enable multi-key transactions, the Xanadu platform was altered instead to include a highly-optimised version of PAXOS. This design choice, bringing with it many useful simplifications, is implemented in a different codebase and the result termed “XanaduPX”.

XanaduPX nodes perform distributed consensus with a similar set of odd-numbered groups of nodes (minimum of 3). Although the PAXOS algorithm also favours one node as the primary gateway to agreeing a new value (i.e. a *leader* node), with modern computer hardware and code optimization it is possible for a single node to manage the full rate of updates without causing the bottlenecks that occur on older hardware (when MultiRaft is the appropriate approach).

XanaduPX nodes also record all updates with one agreement group for the entire key range, and the group nodes keep these 32-byte records in RAM memory until a maximum limit is reached. Unlike XanaduRFT, the XanaduPX implementation stores the log to local disk on the Key-Reference nodes rather than the Xanadu data store and Xanadu Registry nodes. This simplifies both the KV and registry node implementations without introducing any additional failure risks; there are as many permanent disk copies of these logs as there are Paxos nodes[16].

Because the key-reference log is stored on the XanaduPX nodes themselves (during operation in RAM and when shutdown saved to (local) disk), the *XanaduPX* directory is created to hold this data in a set of files as follows (where X is the integer node ID of the paxos node):

- **LogEntries_X.bin** a serialized list of the times (to pseudo nanosecond precision) and data references for the actual data updated at that point in time
- **LogValues_X.bin** contains the data references from above and the raw binary data they refer to. This data is simply the keys and data references (from the full resilient data store) that are altered in this atomic transaction (i.e. the multiple key updates that happen together at the time specified in the LogEntries file).

A key feature of XanaduPX is the multi-key *Atomic, Consistent, Isolated and Durable (ACID)* transactions that it supports. The one PAXOS agreement group covers the entire key range so a transaction covering any number of keys can be submitted to the PAXOS leader and agreement obtained for the whole set of updates in one atomic operation.

[16] By definition there are 2N+1 Paxos nodes if the system architect wishes to enable the system to operate with N failed nodes.

XanaduPX also supports the concept of *long-running* transitions, which all other products struggle with. A long-running transaction is a series of updates that might contain thousands of single key-value updates that take a long time to submit to the consensus leader. During this time a conventional data store must *block* other updates that get submitted while the long-running transaction is uploaded and completed. This "blocking" of updates can be a serious problem for systems having to support high rates of near-simultaneous data updates.

A timeline store like Xanadu presents a simple solution to this problem. When a set of updates is submitted as a transaction an *exclusion region* can be supplied to make the whole transaction a *conditional update*. The condition is that there be no writes to keys in the exclusion region before the conditional update is committed (i.e. the transaction). If another update comes in within the exclusion region before the long-running transaction can complete, Xanadu can fail the long-running transaction immediately, even before all the key-reference updates inside the transaction have been uploaded. Clients that initiated this transaction can then rescan the database and decide whether to retry with another conditional transaction.

Updates that are outside the exclusion region can be completed without blocking, or being blocked by, the long-running transaction. In this way the Xanadu timeline store enables a simple yet critically important operational feature that means transaction size is not limited, either in size or duration. Without limits on transactions, layers on top of the basic Xanadu timeline store can be simplified significantly – in particular the adaptation that provides a standard relational model with SQL query interface is much simpler without limitations on the size of transactions supported by the underlying layer. This is a key differentiator between Xanadu and other products.

3. Store Nodes

The fundamental store nodes receive data block read and write requests from clients, which when including inline deduplication means nodes use the following rules:

1. **Writing data not already stored**: the store node checks for the existence of the data block using the full content hash, which is held in RAM for speed. It copies the data to its local disk, creates a new unique 64-bit *data-reference* (see reference format below). It also forwards the store request to additional storage nodes to satisfy the

clients replication count requirements, and on success returns the data reference to the client.

2. **Writing data already stored**: the store checks for blocks with the same hash code as the incoming block. All matching blocks are then checked in detail (every byte) for a match and if found the store returns the data reference for the existing data block.
3. **Reading data already stored**: the store looks up the data block in its RAM loaded index for the block with the supplied data reference. The most recent block data is often held in RAM too, but if not present it reads this data from disk.

The construction of the data reference for new data ensures global uniqueness and global reuse of the reference if available. To see this, consider the actions of a new client wanting to store a new data block:

1. The client computes the hash code of the block and, using the Xanadu data placement algorithm, the list of "live" storage nodes for this block (a list of all "live" storage nodes can be downloaded from a registry node at any time. A relatively recent copy is sufficient).
2. With a goal of storing N copies of this data block, the client sends the data to the first storage node in the list from (1). If this node is uncontactable, the client connects to the second node and marks the first as "unavailable". Clients rapidly move on to backup nodes as soon as a problem is detected, with automatic recovery as they retry nodes over time.
3. The storage node performs the steps listed in the protocol above; either it finds an existing data block (and reference) or stores a new block and creates a new reference, in either case returning the reference to the client.

All these activities are immune to failures during and after the storage process. If the first storage node fails before completing all the above steps, the client simply restarts the process with another storage node from the list. If the network fails so the client misses the response from the first node, then when it retries it will most likely receive the data reference that the first node created and passed onto other nodes. If the first node has been completely isolated then the client will receive the data reference created by the second node, which is just as valid as the orphaned copy on the first node.

Each Xanadu storage node creates a local directory called *XanaduStore* into which it stores the actual data it is required to. These files are used for the following purposes:

- **XanaduLock** - Facilitates running multiple storage nodes on the same machine without opening the wrong storage file(s) by accident – two stores accessing the same xix, xref and/or xds files would result in corrupted data.
- ***.xix** - Data store index file, contains the externally visible Xanadu data references and an internal index pointing to the actual data stored in the corresponding **xds** file.
- ***.xds** - Data store file, containing the actual data stored with an internal index that is referred to in the corresponding **xix** file.
- ***.xref** - Cross reference file, used by the tablet healer process (see above), where references that have been remapped to allow better deduplication are stored.

The Xanadu data store nodes load the contents of the *xix* files into RAM on start up so that they can quickly find the raw data (via the internal index) within the *xds* file. Remapping, garbage collection and other background operations can result in two or more temporary copies of these files until the tablet healing run completes and then only the primary *xds* and *xix* files are kept. All operations are performed in fail-safe manner that guarantees data cannot get lost if the node crashes in the middle of all these operations.

Data Store Auto-Healing

A close examination of the above storage procedures shows there is a chance of a race-condition where the first node stores the data but fails to contact other nodes to make the replicas or get its reply to the client. While the client simply retries with other nodes, receiving a different data reference from whichever node successfully stores the data, there will now be two or more unique references pointing to identical copies of the data block. Although harmless for data integrity, this is wasteful of storage space.

To address this issue, Xanadu Store nodes implement a data “healing” process. This background process periodically scans the contents of each node’s index and looks for blocks of data that are duplicates. If one is found, the copy with the data reference with the lowest nodeID is preferred and data blocks pointed to by other references are replaced with simple (internal) redirections to this primary copy. Note these duplicate data references are never deleted because some clients will have been sent these as valid ways to retrieve data blocks, which the storage nodes must honour.

Another situation where additional space could be used unnecessarily is when, due to a primary storage node being down for a period, clients make copies of the data on nodes that are higher in the node list than would be addressed if the first node had been available. For example, consider the following ordered list of nodes for a data block, as calculated by the Xanadu data placement algorithm:

[1,56,23,42,3,21,27,43….]

If a client requires 3 redundant copies of the data, then it will attempt to store the data on nodes 1, 56 and 23. If node 1 is unavailable the client will store the copies on nodes 56, 23 and 42. When node 1 comes back online the self-healing process on nodes 56, 23 and 42 will all note that a copy should be on node 1 and send a copy there[17]. The same healing process on node 42 will then note that there are now 3 redundant copies on nodes 1,56 and 23 and delete its copy.

The healing process also acts to move data around nodes when additional nodes are added or taken away (permanently). An essential part of horizontal scalability is the ability to add new nodes during operation without noticeable downtime. In Xanadu, the addition of a node means the appearance of a new node ID in the ordered storage lists like the one above, and hence the position of a whole range of data blocks becomes slightly incorrect according to the rule that requires placement on the first N nodes in the list (where N is the number of replicas required).

The self-healing processes on each node serve to move blocks around to even out the storage load – making new copies on the added nodes as well as deleting them from other nodes and thereby freeing up space. The healers also act when nodes are removed, rebalancing data to again maintain the correct number of data block copies.

[17] There is a randomised time delay built in to each self-healing process so not all three nodes will attempt to make these repairs at the same time. As a result one of them will eventually succeed, the others will see the new copy in the right place and not send data around unnecessarily.

4. Xanadu Clients (APIs)

Clients are the application programs that use the Xanadu API and library to connect to a running Xanadu instance. The Xanadu API contains a core set of calls that act as the foundation for increasingly complex applications. The Xanadu connection is provided to client software inside a standard Java archive file XanaduPX.jar or XanaduRFT.jar, often just referred to as *drivers.*

Client software loads the Xanadu drivers and initializes connections using the *XanaduConnectionFactory* class. This connection factory provides simple methods to open a connection to Xanadu's DataStore and KeyValueStore. The Xanadu drivers initiate the connection(s) to the Xanadu registry and have a local implementation of the node placement algorithm so that data storage requests can be made directly to all storage nodes without any network bottlenecks.

Xanadu Data Store API

Created by a call to the XanaduConnectionFactory, the Data Store is the raw horizontally scalable content addressable deduplicating storage system. Clients store data via the Xanadu drivers by calling:

```
long dataRef = storeData(long length, InputStream source)
```

The input data stream *source* is read for *length* bytes and a unique data reference is returned. Similarly, a client can read stored data back by supplying this data reference to:

```
InputStream data = readData(long dataReference)
```

The client then reads the data via the standard InputStream interface.

Useful Data Store APIs are:

DataBlockStore = XanaduConnectionFactory. createDataBlockStore(java.net.InetSocketAddress[], xanaduRegistryAddresses, DataStoreURI storeURI, int replicas, Boolean useChecksums)
Creates a new DataBlockStore given the Xanadu registry address.

DataStore = XanaduConnectionFactory. createDataStore(DataBlockStore ds, int blockSize)
Creates a new DataStore given the underlying DataBlockStore

DataStore.storeData(long length, java.io.InputStream src)
DataStore.readData(long dataReference)
DataStore.getDataSize(long dataReference)
DataStore.close()

DataBlockStore.readDataBlock(long dataReference, byte[] buffer, int offset, int length)
DataBlockStore.storeDataBlock(byte[] buffer, int offset, int length)
DataBlockStore.lookupDataBlock(byte[] buffer, int offset, int length)

Xanadu Key Reference Store API

The Key-Reference store, termed simply the *Key-Value* store in XanaduRFT and XanaduPX, is the implementation of the distributed agreement system that updates the values of keys and the values they have at any time (i.e. the *Timelines*). Following Xanadu's two-layer storage model, keys (i.e. Strings/Text labels) are set to data references as supplied by the data store, rather than the raw data bytes themselves.

The basic update method is:

```
long timeNS = setValue(String key, long timeHintNS, DataStoreURI storeURI, long dataStoreReference);
```

Where the timeHintNS is a suggested time for the atomic update (e.g. current system time in nano seconds which can be obtained by calling a Xanadu API ReferenceTools.nowNanos()) – the method returns the globally unique timestamp assigned to the update (which may differ from the time hint for internal reasons).

To retrieve a key's value within a time range, defined as the most recent value set before endTimeNanos but not earlier than startTimeNanos:

```
KeyValue kv = getValueBefore(String key, long startTimeNanos, long endTimeNanos);
```

The *endTimeNanos* can be specified as -1, in which case the search range is for all time into the future. Similarly, *startTimeNanos* can be -1, implying the search range is for all time in the past. A similar call retrieves a key's value as at or after startTimeNanos, but ignoring any values set after endTimeNanos:

```
KeyValue kv = getValueAfter(String key, long startTimeNanos, long endTimeNanos);
```

There also methods to access various collections of keys and their values in time ranges. The values of a key over a time range can be queried using the following call:

```
Constant KeyValueIterator kvs = getKeyValues(String key, boolean ascending, long startTimeNanos, long endTimeNanos);
```

A list of keys defined at a specific time is returned by:

```
KeyIterator itt = getKeys(long minTimeNanos, long maxTimeNanos);
```

and for all keys and values in a time range:

```
KeyValueIterator itt=getAllValues(long startTimeNanos, long endTimeNanos);
```

Additional useful Key Reference Store APIs are:

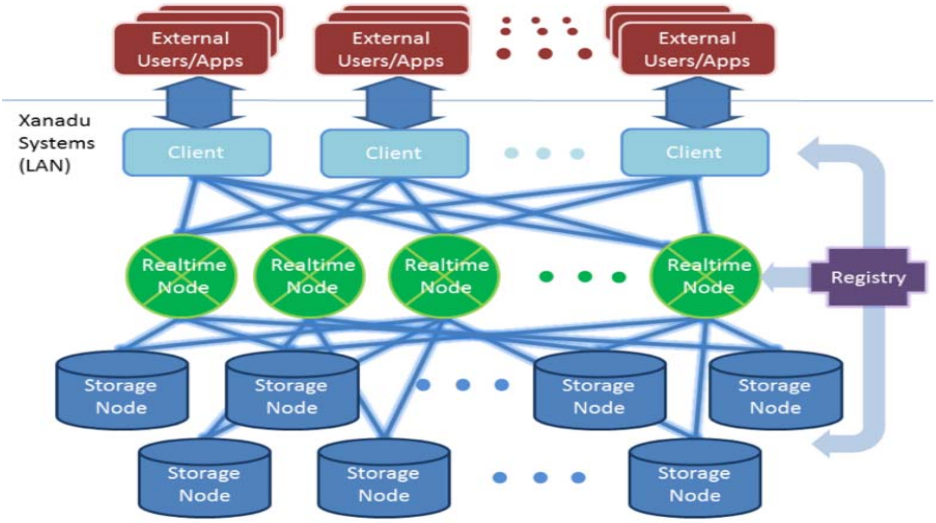
KeyValueStore=XanaduConnectionFactory.
createKeyValueStore(java.net.InetSocketAddress[] xanaduRegistryAddresses)
A convenience method that uses the default client cache directory to cache results.

KeyValueStore.setValues(SetValueTask[] tasks, int offset, int length)
A convenience method for multiple calls of the 'setValue' method, which may be more efficient than looping around multiple separate calls. Note this call does NOT offer transactional guarantees, and is consequently faster than an equivalent 'setValuesAtomic' call.
KeyValueStore. getTimeMillis()
Returns the time in milliseconds at which this value was applied to this key

III. Installation

Xanadu runs on any Java 1.8 Virtual Machine (JVM). Free versions are available from Oracle for all versions of Windows, Mac and Linux. Xanadu works with OpenJDK but is not supported on this platform.

Xanadu is a distributed system, so multiple processes are needed to bring a system fully online. For testing purposes, it is common to launch these processes on a single computer. In production Xanadu derives is resilience (i.e. no data loss or service interruptions from failed nodes or broken network connections) from the assumption that nodes fail randomly and independently. Production systems therefore should be run on separate computers to avoid the possibility of correlated failures. A system and network diagram of a Xanadu production deployment is shown below:



The key components of any Xanadu system are the *registry nodes, realtime nodes,* and *storage nodes.* Xanadu clients are applications that include the XanaduRFT *driver* software which handles the connection to the other nodes and exposes the *Xanadu API* to application code.

1. Registry

The core details of any Xanadu instance are held by the registry nodes. There can be multiple registry nodes according to the resilience requirements, and a Xanadu system is considered available if even one registry node is available. For basic resilience use two registry nodes, three or four should be sufficient for “five 9’s” service levels.

Start a registry node with the following command via the Linux/Mac command shell or Windows command prompt:

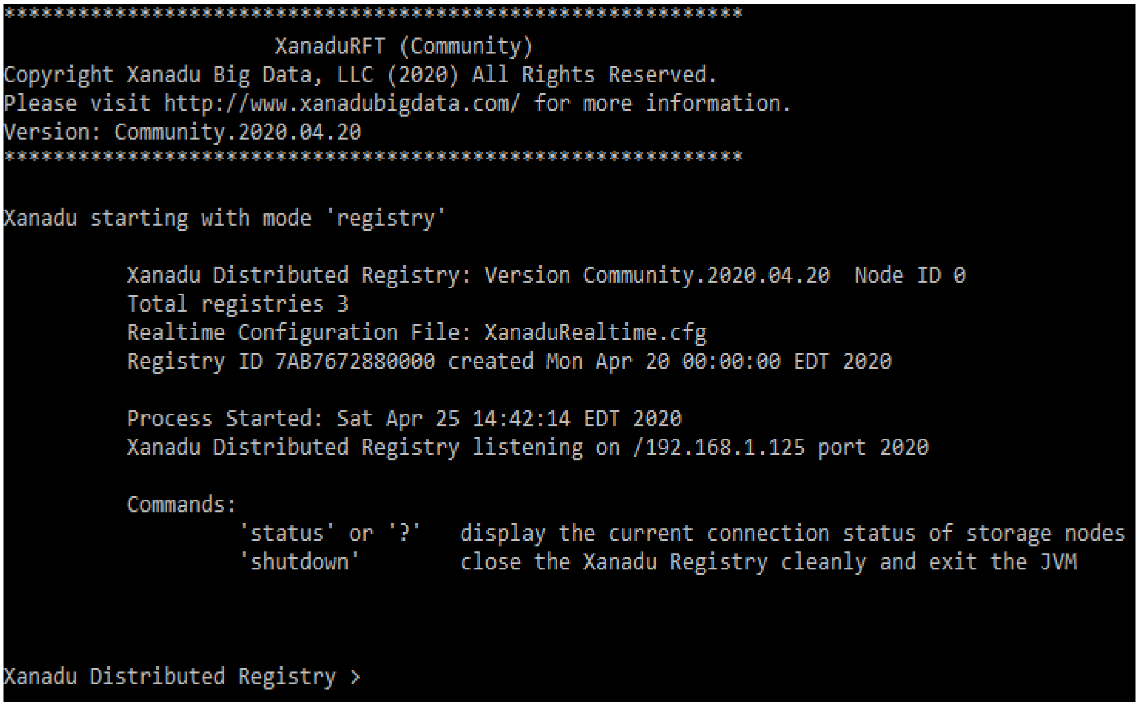
```
java -jar XanaduRFT.jar -mode registry –registryNodeNumber 0
```

The *registryNodeNumber* is a numeric identifier starting from zero which must be unique for each resilient registry node. Other configurable parameters are contained in two files that should be in the same directory as the XanaduRFT.jar file when running as a registry:

1. config.js
2. XanaduRealtime.cfg

This config.js must be copied to all nodes and clients so they are aware of where the registries for a particular Xanadu instance are on the network. As the registries supply all but the basic details (contained in the config.js file) to Storage nodes and clients, the XanaduRealtime.cfg file is only required to be in the same directory as the processes that run the Xanadu registries.

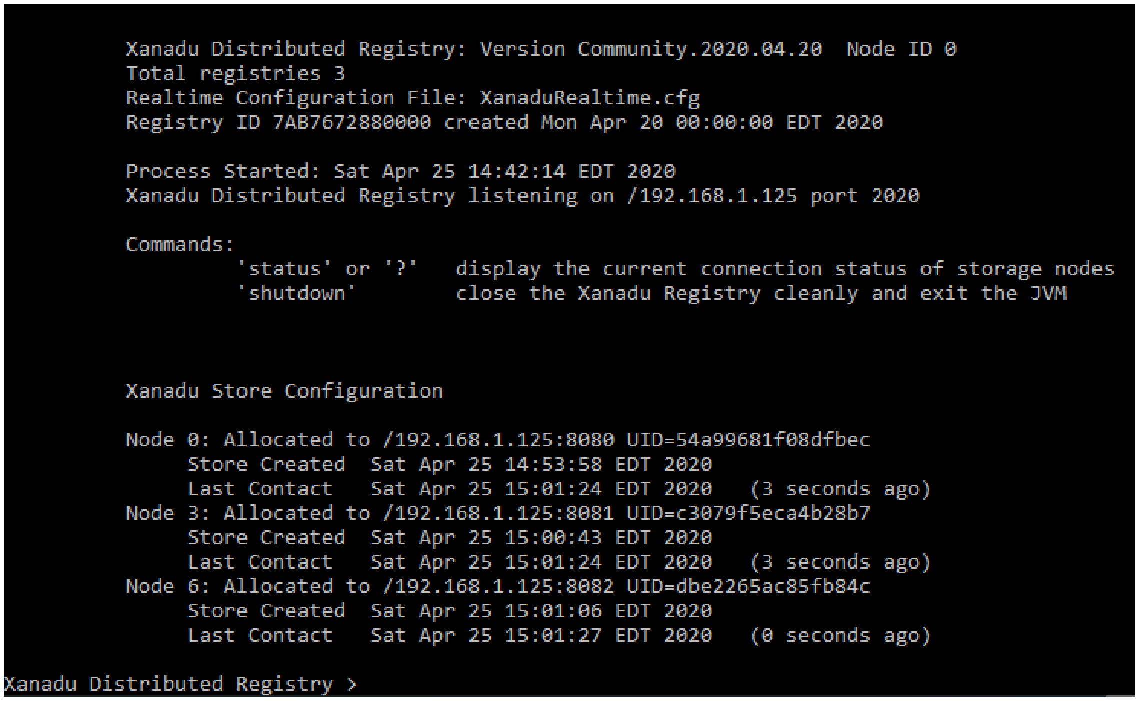
When started, the registry node prints out a range of useful information pertaining to the installation:



```
************************************************************
                      XanaduRFT (Community)
Copyright Xanadu Big Data, LLC (2020) All Rights Reserved.
Please visit http://www.xanadubigdata.com/ for more information.
Version: Community.2020.04.20
************************************************************

Xanadu starting with mode 'registry'

          Xanadu Distributed Registry: Version Community.2020.04.20  Node ID 0
          Total registries 3
          Realtime Configuration File: XanaduRealtime.cfg
          Registry ID 7AB7672880000 created Mon Apr 20 00:00:00 EDT 2020

          Process Started: Sat Apr 25 14:42:14 EDT 2020
          Xanadu Distributed Registry listening on /192.168.1.125 port 2020

          Commands:
                   'status' or '?'   display the current connection status of storage nodes
                   'shutdown'        close the Xanadu Registry cleanly and exit the JVM


Xanadu Distributed Registry >
```

The current state of the registry and the known storage nodes connected to it will be displayed by pressing <return> on this console window (see below).

The best way to shut the registry is to type “shutdown” on this console window to let the system save state cleanly.



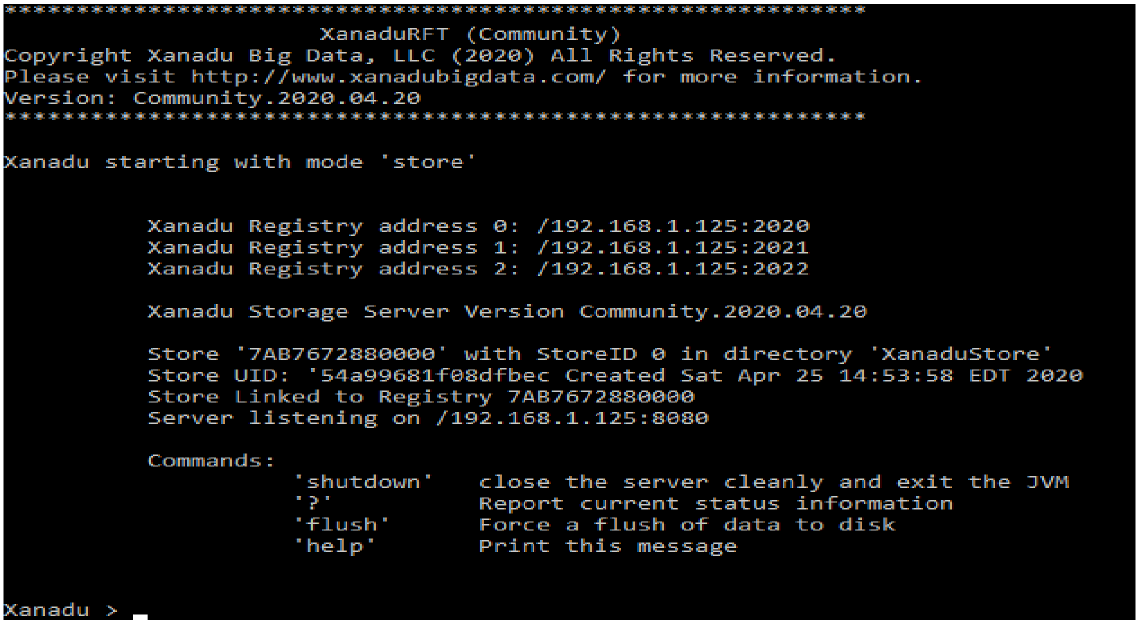
```
          Xanadu Distributed Registry: Version Community.2020.04.20  Node ID 0
          Total registries 3
          Realtime Configuration File: XanaduRealtime.cfg
          Registry ID 7AB7672880000 created Mon Apr 20 00:00:00 EDT 2020

          Process Started: Sat Apr 25 14:42:14 EDT 2020
          Xanadu Distributed Registry listening on /192.168.1.125 port 2020

          Commands:
                   'status' or '?'   display the current connection status of storage nodes
                   'shutdown'        close the Xanadu Registry cleanly and exit the JVM


          Xanadu Store Configuration

          Node 0: Allocated to /192.168.1.125:8080 UID=54a99681f08dfbec
               Store Created  Sat Apr 25 14:53:58 EDT 2020
               Last Contact   Sat Apr 25 15:01:24 EDT 2020   (3 seconds ago)
          Node 3: Allocated to /192.168.1.125:8081 UID=c3079f5eca4b28b7
               Store Created  Sat Apr 25 15:00:43 EDT 2020
               Last Contact   Sat Apr 25 15:01:24 EDT 2020   (3 seconds ago)
          Node 6: Allocated to /192.168.1.125:8082 UID=dbe2265ac85fb84c
               Store Created  Sat Apr 25 15:01:06 EDT 2020
               Last Contact   Sat Apr 25 15:01:27 EDT 2020   (0 seconds ago)

Xanadu Distributed Registry >
```

2. Storage Nodes

Xanadu storage nodes configure themselves once supplied with the registry addresses, and make their local disk space available for deduplicated, parallel data storage. Assuming the config.js file is available in the local directory, in which the registry addresses are specified as described above, a storage node is started with the following command:

```
java -jar XanaduRFT.jar -mode store -listenAddress 202.3.4.5:8080
```

The storage node prints out diagnostic information when it starts, and by pressing<*enter*> on the console the latest information about its internal state is displayed (see below):



```
************************************************************
                      XanaduRFT (Community)
Copyright Xanadu Big Data, LLC (2020) All Rights Reserved.
Please visit http://www.xanadubigdata.com/ for more information.
Version: Community.2020.04.20
************************************************************

Xanadu starting with mode 'store'


          Xanadu Registry address 0: /192.168.1.125:2020
          Xanadu Registry address 1: /192.168.1.125:2021
          Xanadu Registry address 2: /192.168.1.125:2022

          Xanadu Storage Server Version Community.2020.04.20

          Store '7AB7672880000' with StoreID 0 in directory 'XanaduStore'
          Store UID: '54a99681f08dfbec Created Sat Apr 25 14:53:58 EDT 2020
          Store Linked to Registry 7AB7672880000
          Server listening on /192.168.1.125:8080

          Commands:
                    'shutdown'   close the server cleanly and exit the JVM
                    '?'          Report current status information
                    'flush'      Force a flush of data to disk
                    'help'       Print this message


Xanadu > _
```

Again the node can be shutdown cleanly (flushing all pending writes to disk) by typing the *shutdown* command into the console.

A sudden node failure is handled by XanaduRFT because all data is written to more than one storage node before clients are advised of success. Clients can also retrieve the data from more than one storage node, so the absence of a single storagenode does not affect Xanadu's overall availability. With a higher number of replicas stored, data remains available when more than one storage node is unavailable.

3. Realtime Nodes

The XanaduRFT realtime nodes implement the RAFT distributed agreement algorithm[18]. These nodes collectively agree the updates/edits to the key/values stored in Xanadu in a way that is resilient to node failures and network partitions. RAFT uses majority voting to ensure resilience, so there must be at least 3 realtime nodes to ensure resilience to one failure. A larger odd number of realtime nodes can be installed to allow for more than one concurrent failure.

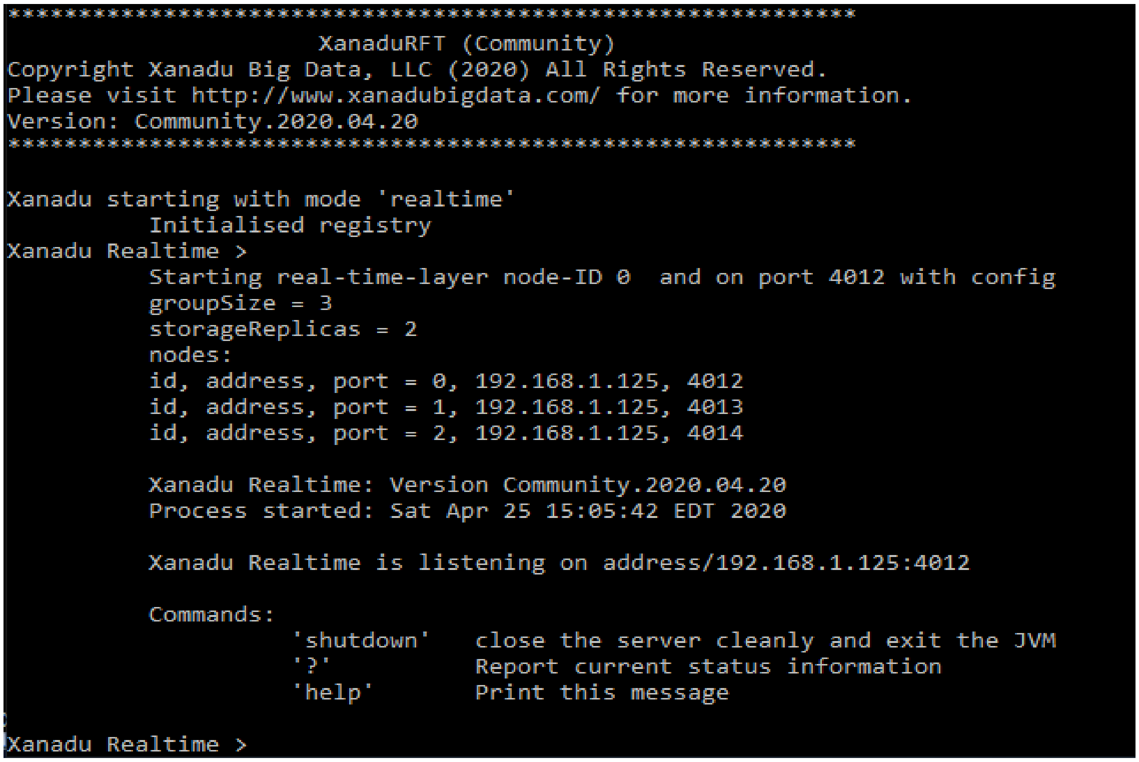
The file config.js must be in the directory where the realtime node(s) are started. These can be launched with the command:

[18] https://raft.github.io/

java -jar XanaduRFT.jar -mode realtime -listenAddress 192.168.1.2

Each node started in this way will examine the IP addresses listed in the XanaduRealtime.cfg (as sent to it from the registry) and bind to the first one it is able to. For separate machines, ensure the IP address of the machine is listed specifically in the config file (rather than simply 0.0.0.0)[19]. All IP addresses in this config file are compared to the *listenAddress* parameter above and the first matching one that the process successfully opens defines its node ID (and thus its role in the agreement group).

When a realtime node starts it prints the following information to the console:



```
************************************************************
                      XanaduRFT (Community)
Copyright Xanadu Big Data, LLC (2020) All Rights Reserved.
Please visit http://www.xanadubigdata.com/ for more information.
Version: Community.2020.04.20
************************************************************

Xanadu starting with mode 'realtime'
          Initialised registry
Xanadu Realtime >
          Starting real-time-layer node-ID 0  and on port 4012 with config
          groupSize = 3
          storageReplicas = 2
          nodes:
          id, address, port = 0, 192.168.1.125, 4012
          id, address, port = 1, 192.168.1.125, 4013
          id, address, port = 2, 192.168.1.125, 4014

          Xanadu Realtime: Version Community.2020.04.20
          Process started: Sat Apr 25 15:05:42 EDT 2020

          Xanadu Realtime is listening on address/192.168.1.125:4012

          Commands:
                    'shutdown'   close the server cleanly and exit the JVM
                    '?'          Report current status information
                    'help'       Print this message

Xanadu Realtime >
```

Note once again that the realtime nodes can be shut down cleanly by entering the “shutdown” command on the console. If any one node is closed in this way all the other realtime nodes will shutdown together after each sending their “KeyLog” (a record of all edits to the keys) to the registry nodes for permanent storage. If a realtime node is closed by force (e.g. <cntrl>-c) the other nodes will continue if possible (i.e. only 1 of 3 have closed, leaving a majority

[19] Except for testing purposes, when all nodes run as separate processes on the same machine, in which case the localhost address (with different ports) can be specified as “0.0.0.0:3000” etc.

working). The closed node can be restarted at any time and it will re-join the group automatically.

As the realtime nodes do not use their local storage (disk) for any important information – all Xanadu data is written to the data storage nodes (see below) or the registries – there is no need to perform backups of the realtime nodes.

IV Tutorial

Tutorial is split into two parts, fully shown in "KeyValueDataStoreTutorialA.java" and "KeyValueDataStoreTutorialB.java". In the first part we update and then query a key-value entry in the Xanadu Key-Value Store and associate it with a data-reference in the Xanadu Data Store. In the second part, we use the bulk-set methods of the Key-Value Store, query it for keys that have been updated in a time range, and query the Key-Value Store to iterate through key-value updates made to those keys in the same time period.

1. Tutorial A

Lets run through the code in " KeyValueDataStoreTutorialA". First, we use the XanaduConnectionFactory to create a connection to the Xanadu Key-Value Store and Data Store:

```
...
private static final DataStoreURI storeURI = new DataStoreURI("TestKVDS");
…
private static DataStore dataStore;
private static DataBlockStore dbs;
private static KeyValueStore kvs;
private static InetSocketAddress[] registryAddresses;
…
registryAddresses = cfg.getIPAddresses("registries",
DistributedRegistry.DEFAULT_PORT);
Replicas = cfg.getInt("replicas", numReplicas);
blockSize = cfg.getInt("blocksize", block_Size);
blockSize = blockSize * BYTES_IN_MB;
...
kvs = XanaduConnectionFactory.createKeyValueStore(registryAddresses);
```

```
dbs = XanaduConnectionFactory.createDataBlockStore(registryAddresses, storeURI,
Replicas, use_Checksums);
dataStore = XanaduConnectionFactory.createDataStore(dbs, blockSize);
...
```

The Xanadu Key-Value Store and Data Store connection requires the Internet Address and port on which each Xanadu Registry process is listening (registryAddresses). The client then queries the Registry to find out where the Xanadu Key-Value Store (KeyValueStore kvs) and Data Store (DataStore dataStore; DataBlockStore dbs) nodes are running and connects directly to them over TCP.

Let's run through this code:

- storeURI : the Data Store URI under which this data will be stored.
- Replicas : the number of replicas all data will be stored under using this connection. For example, if Replicas is set to 3, then any data stored will be stored in three places, by three different Xanadu data store processes in the cluster. It should be noted that to successfully store any data you should specify a value less than or equal to the number of nodes in your data store cluster. The maximum replica count possible is 3 within the Community version, but 8 in the full version.
- use_Checksum : whether or not the client should add an additional integrity check on any data to be stored before sending it to the data store nodes. For safety this should be set to true.
- blockSize : Xanadu automatically splits data into chunks if the data is bigger than the blockSize (by default, 64MB in size). In the Community version the maximum size of a single block of data is 64MB, while in the full version there is no inbuilt limit.

Next, we will store some String message in the Data Store for each update.

```
String keyName = "TestKeySetA";
String msg = "TestDataSetA";
int numUpdates = 10;
...
byte[] storedData = ResourceUtils.getBytes(msg);
long reference = dataStore.storeData(storedData.length, new
ByteArrayInputStream(storedData));
long currentTimeNanos = ReferenceTools.nowNanos();
long storeTimeNanos = kvs.setValue(keyName, currentTimeNanos, storeURI, reference);
```

```
...
long testReference = dbs.lookupDataBlock(storedData, 0, storedData.length);
...
```

First we define a String message to be stored: msg. Next, we convert that String to an array of bytes to be stored. Then, the Data Store connection stores the data. Finally, the Key-Value Store is updated.

We can now query the Key-Value store for the updated key value (kv) using either known updated time or key name and by calling the KeyValueStore (kvs) method either getKeys/getKeyValues or getValueBefore respectively.

```
...
long startupdateTimeNanos = Knowntime[0];
long endupdateTimeNanos = Knowntime[numUpdates-1];
KeyIterator kit = kvs.getKeys(startupdateTimeNanos-1, endupdateTimeNanos+1);
...
ConstantKeyValueIterator itt = kvs.getKeyValues(key, startupdateTimeNanos-1, endupdateTimeNanos+1);
...
while ((kv = itt.next()) != null)
...
```

Or,

```
...
long startTime = 0;
long finishTime = Long.MAX_VALUE;
...
KeyValue kv = kvs.getValueBefore(Knownkey[i], startTime, finishTime);
...
```

Having successfully stored the data, we will now use the same connection to retrieve the data and compare it with what we asked to be stored.

```
long timeAdded = kv.getTimeMillis();
long retrievedReference = kv.reference;
...
InputStream data = dataStore.readData(retrievedReference);
...
```

Finally, we close the connection to the stores. This frees up any resources it has created in the heap and closes any all connections to the store cluster.

Once you setup the store cluster (e.g. 3 registries, 3 store nodes, 3 realtime nodes), you can execute this tutorial with the commands:

```
javac -cp XanaduRFT.jar;. (or :.) KeyValueDataStoreTutorialA.java
java -cp XanaduRFT.jar;. (or :.) KeyValueDataStoreTutorialA (options: -replicas 2 -blocksize 1 -registries IP1:port1,IP2:port2,IP3:port3)
```

2. Tutorial B

We now move on to the KeyValueDataStoreTutorialB.java. In this section, we will make bulk updates to both the Key-Value Store. This is more efficient (in terms of network bandwidth between the client and the cluster) than making individual updates and also will allow us to do more advanced queries.

The code in KeyValueDataStoreTutorialB.java begins in the same way as before, setting up client connections to the Xanadu Key-Value Store and Data Store. Next we make our bulk inserts.

```
...
SetValueTask[] tasks = new SetValueTask[numUpdates];
...
tasks[i] = new SetValueTask(key, currentTimeNanos, storeURI, references);
...
kvs.setValues(tasks, 0, numUpdates);
...
```

Lets go through this line-by-line. First we store our multiple data inputs and an array of SetValueTask objects (tasks[i]) for the stored multiple data inputs. Xanadu SetValueTask objects act like JDBC ResultSet object, but each object specifies an individual key-value store update we wish to make. Next, we make our bulk Key-Value Store update by calling the KeyValueStore (kvs) method setValues. This has the exactly the same effect as if we called KeyValueStore.setValue(..) in a loop for each update, but is more efficient. After calling this method, the SetValueTasks we passed into this method have been updated.

We can now query the Key-Value store for the key-value updates using SetValueTask objects (tasks[i]).

```
...
Knowntime[i] = tasks[i].getNanoTimeResult();
...
knownKeyValues.add(tasks[i].getKeyValue());
...
```

Once you setup the store cluster (e.g. 3 registries, 3 store nodes, 3 realtime nodes), you can execute this tutorial with the commands:

```
javac -cp XanaduRFT.jar;. (or :.) KeyValueDataStoreTutorialB.java
java -cp XanaduRFT.jar;. (or :.) KeyValueDataStoreTutorialB (options: -replicas 2 -blocksize 1 -registries IP1:port1,IP2:port2,IP3:port3)
```

If you have any questions
please contact Alex G. Lee
at alexglee@xanadubigdata.com