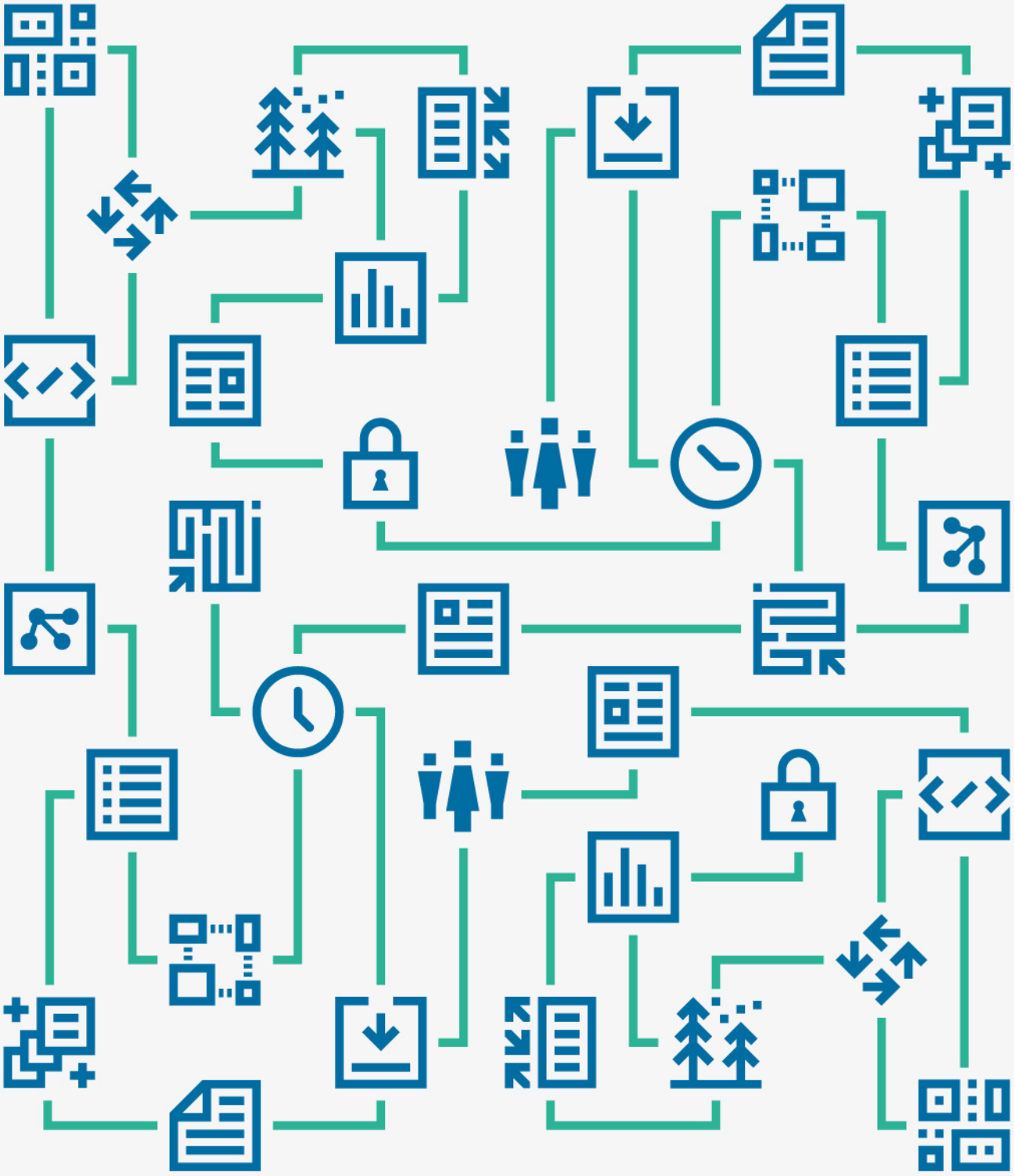
High-availability Manual

Sparksee by Sparsity Technologies



Contents

Introduction

As of version 4.7 SPARKSEE high-performance graph database comes with high-availability features which are best suited for those applications with large petition load.

In fact, SPARKSEE high-availability (SPARKSEEHA) enables multiple replicas working together, allowing the highest scalability for SPARKSEE applications.

This document covers the architecture for these SPARKSEEHA features, the configuration details for enabling it, and examples of typical usage scenarios.

This first version of SPARKSEEHA allows to horizontal scaling of read operations whilst writes are managed centrally. Future work on SPARKSEEHA will provide fault tolerance and master re-election.

SPARKSEEHA is a software feature which is enabled through the license. SPARKSEE free evaluation does not provide it by default. More information about the licenses at Sparksee website.

Architecture

Design

SPARKSEEHA provides a horizontally scaling architecture that allows SPARKSEE-based applications to handle larger read-mostly workloads.

SPARKSEEHA has been thought to minimize developers' work to go from a single node installation to a multiple node HA-enabled installation. In fact, it does not require any change in the user application because it is simply a question of configuration.

To achieve this, several SPARKSEE slave databases work as replicas of a single SPARKSEE master database, as seen in the figure below. Thus, read operations can be performed locally on each node and write operations are replicated and synchronized through the master.

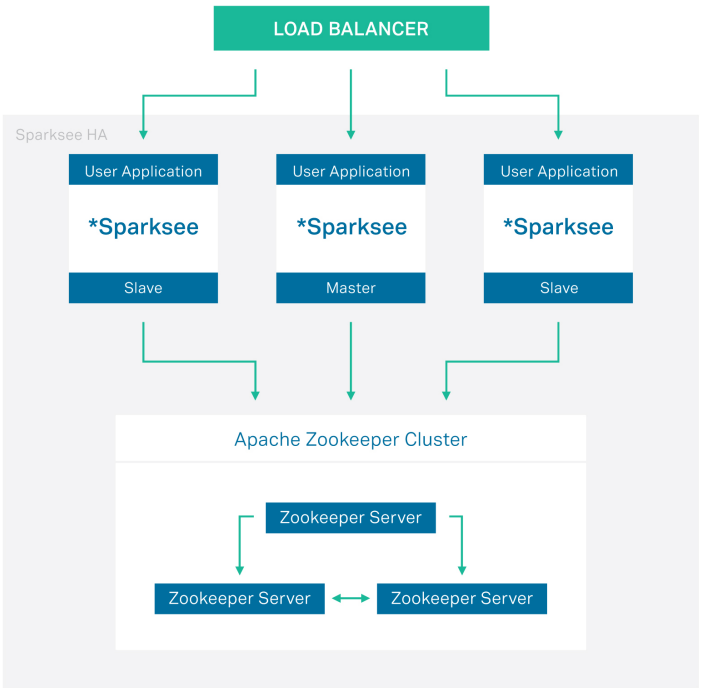


Figure 1.1: SPARKSEEHA Architecture

Figure 1.1 shows all components in a basic SPARKSEEHA installation:

- **SPARKSEE master**

  This is responsible for receiving write requests from a slave and redirecting them to the other slave instances. At the same time the master itself also plays the role of a slave.

Only a single node of the cluster can be configured to be the master. The election of the master is automatically done by the coordinator service when the system starts.

The master is in charge of the synchronization of write operations with the slaves. To do this task it manages a history log where all writes are serialized. The size of this log is limited, and it can be configured by the user.

- **SPARKSEE slave**

  Slaves are exact replicas of the master database; they can therefore locally perform read operations on their own without requiring synchronization.

  However, for write operations the synchronization with the master in order to preserve data consistency is a must. These writes are eventually propagated from the master to other slaves. Therefore, the result of a write operation is not immediately visible in all slaves. These synchronizations are made by default during the next write operation; however there is an optional polling to force synchronization that can be configured by the user.

  It is not mandatory to have a slave in the architecture, as the master can work as a standalone.

- **Coordinator service: Apache ZooKeeper**

  A ZooKeeper cluster is required to perform the coordination tasks, such as the election of the master when the system starts.

  The size of the ZooKeeper cluster depends on the number of SPARKSEE instances. In every case, the size of the ZooKeeper cluster must be an odd number.

  SPARKSEE v4.7 works with Apache ZooKeeper v3.4. All our tests have been performed using v3.4.3.

- **User application**

  As SPARKSEE is an embedded graph database, a user application is required for each instance. As it has already been mentioned, moving to SPARKSEEHA mode does not require any update in the user application.

  Note that the user application can be developed for all the platforms and languages supported by SPARKSEE. For the current version will be running on Windows, Linux or MacOSX and using Java, .NET or C++

- **Load balancer**

  The load balancer redirects the requests to each of the running applications (instances).

  The load balancer is not part of the SPARKSEE technology, therefore it must be provided by the user.

  In order to achieve the horizontal scalability, this redistribution of the application requests must be done efficiently. A round-robin approach would be a good starting solution but depending on the application requirements smarter solutions may be required. In fact, using existing third-party solutions is advisable.

More information about load balancing strategies & available solutions in this article.

How it works

Now that the pieces of the architecture are clear, let's see how SPARKSEEHA works in different scenarios or acts in typical operations using these components. Below is an explanation of how the system acts in the described situations.

Master election

The first time a SPARKSEE instance goes up, it registers itself into the coordinator service. The first instance registered which becomes the *master*. If a master already exists, it becomes a *slave*.

Reads

As all SPARKSEE slave databases are replicas of the SPARKSEE master database, slaves can answer read operations by performing the operation locally. They do not need to synchronize with the master.

Writes

In order to preserve data consistency, write operations require slaves to be synchronized with the master. A write operation is as follows:

1. A slave wishes to perform a write operation and sends it to the master.
2. The master serializes the operation in the history log, performs the write, and replies to the slave when it has been successfully achieved.
3. From the master the slave receives a fully updated list of write operations, which are extracted from the history log, and records them in addition to its original write. This operation preserves the consistency of the database.

If two slaves perform a write operation on the same object at the same time, it may result in a *lost update* in the same way as may happen in a SPARKSEE single instance installation if two different sessions want to write the same object at the same time.

Slave goes down

A failure in a slave during a regular situation does not affect the rest of the system. However if it goes down in the middle of a write operation the behavior of the rest of the system will depend on the use of transactions:

- If we are in an **auto-committed** mode (the user does not explicitly start/end transactions), the system remains operational when the slave fails.
- If the write operation **is enclosed within a transaction**, the master will not be able to *rollback* the operation and will therefore be blocked for any more write operations. This behavior can be explained by the fact that the master keeps waiting to finish the transaction, which will never be received as the slave has crashed.

Slave goes up

When a SPARKSEE instance goes up, it registers itself with the coordinator. The instance will become a slave if there is already a master in the cluster.

If **polling** is **enabled** for the slave, it will immediately synchronize with the master to receive all pending writes. On the other hand, if **polling** is **disabled**, the slave will synchronize when a write is requested (as explained previously).

Future work

This is a first version of SPARKSEEHA, so although it is fully operational some important functionality is not available which will assure a complete high-availability of the system. Subsequent versions will focus on the following features:

Master goes down

A failure in the master leaves the system non-operational. In future versions this scenario will be correctly handled automatically converting one of the slaves into a master.

Fault tolerance

A failure during the synchronization of a write operation between a master and a slave leaves the system non-operational. For instance, a slave could fail during the performance of a write operation enclosed in a transaction, or there could be a general network error.

This scenario requires that the master should be able to abort (*rollback*) a transaction. As SPARKSEE does not offer that functionality, these scenarios cannot currently be solved. SPARKSEEHA will be able to react when SPARKSEE implements the required functionality.

Configuration

Installation

A complete installation includes all the elements previously described in the architecture: SPARKSEE (SPARKSEEHA configuration), the coordination service (ZooKeeper) and the load balancer. The last one byond the scope of this document because, as has been previously stated, it is developers' decision which is the best to use for their specific system.

SPARKSEEHA is included in all distributed SPARKSEE packages. Thus, it is not necessary to install any extra package to make the application HA-enabled it is only a matter of configuration. SPARKSEE can be downloaded as usual from Sparsity's website. Use SPARKSEE to develop your application. Plus, visit SPARKSEE documentation site to learn how to use SPARKSEE.

SPARKSEEHA requires Apache ZooKeeper as the coordination service. Latest version of ZooKeeper v3.4.3 should be downloaded from their website. Once downloaded, it must be installed on all the nodes of the cluster where the coordination service will run. Please note that Apache ZooKeeper requires Java to work, we recommend consulting the Apache ZooKeeper documentation for requirements details.

ZooKeeper

The configuration of Apache ZooKeeper can be a complex task, so we refer the user to the Apache ZooKeeper documentation for more detailed instructions.

This section does, however, cover the configuration of the basic parameters to be used with SPARKSEEHA, to serve as an introduction for the configuration of the ZooKeeper.

Basic ZooKeeper configuration can be performed in the $ZOOKEEPER_HOME /conf/zoo.cfg file. This configuration file must be installed on each of the nodes which is part of the coordination cluster.

- clientPort

  This is the port that listens for client connections, to which the clients attempt to connect.

- dataDir

  This shows the location where ZooKeeper will store the in-memory database snapshots and, unless otherwise specified, the transaction log of updates to the database. Please be aware that the device where the log is located strongly affects the performance. A dedicated transaction log device is a key to a consistently good performance.

- tickTime

  The length of a single tick, which is the basic time unit used by ZooKeeper, as measured in milliseconds. It is used to regulate heartbeats, and timeouts. For example, the minimum session timeout will be two ticks.

- server.x=[hostname]:nnnnn[:nnnnn]

  There must be one parameter of this type for each server in the ZooKeeper ensemble. When a server goes up, it determines which server number it is by looking for the myid file in the data directory. This file contains the server number in ASCII, and should match the x in server.x of this setting. Please take into account the fact that the list of ZooKeeper servers used by the clients must exactly match the list in each one of the Zookeper servers.

  For each server there are two port numbers nnnnn. The first port is mandatory because it is used for the Zookeeper servers, assigned as followers, to connect to the leader. However, the second one is only used when the leader election algorithm requires it. To test multiple servers on a single machine, different ports should be used for each server.

This is an example of a valid $ZOOKEEPER_HOME/conf/zoo.cfg configuration file:

```
tickTime=2000
dataDir=/var/lib/zookeeper/
clientPort=2181
initLimit=5
syncLimit=2
server.1=zoo1:2888:3888
server.2=zoo2:2888:3888
server.3=zoo3:2888:3888
```

SPARKSEEHA

As previously explained, enabling HA in a SPARKSEE-based application does not require any update of the user's application nor the use of any extra packages. Instead, just a few variables must be defined in the SPARKSEE configuration.

- sparksee.ha

  Enables or disables HA mode.

  Default value: false

- sparksee.ha.ip

  IP address and port for the instance. This must be given as follows: ip:port. It follows the same policy as Zookeeper, the user must configure it in order to be able to use a private or public IP.

  Default value: localhost:7777

- sparksee.ha.coordinators

  Comma-separated list of the ZooKeeper instances. For each configuration file in an instance, the IP address and the port must be given as follows: ip:port. Moreover, the port must correspond to that given as clientPort in the ZooKeeper configuration file. There is no need to have

a Zookeeper instance per DEX instance, in fact for a small architecture very few Zookeper servers are enough.

Default value: “”

- sparksee.ha.sync

  Synchronization polling time. If 0, polling is disabled and synchronization is only performed when the slave receives a write request, otherwise the parameter fixes the frequency the slaves poll the master asking for writes. The polling timer is reset if the slave receives a write request, at that moment it is (once again) synchronized.

  The time is given in *time-units*: <X>[D|H|M|S|s|m|u] where <X> is a number followed by an optional character representing the unit: D for days, H for hours, M for minutes, S or s for seconds, m for milliseconds and u for microseconds. If no unit character is given, seconds are assumed.

  Default value: 0

- sparksee.ha.master.history

  The history log is limited to a certain period of time, so writes occurring after that period of time will be removed and the master will not accept requests from those deleted SPARKSEE slaves.

  For example, in case of 12H, the master will store in the history log all write operations performed during the previous 12 hours. It will reject requests from a slave which has not been updated in the last 12 hours.

  This time is given in *time-units*, as with the previous variable.

  Default value: 1D

Please, take into account the fact that slaves should synchronize before the master's history log expires. This will happen if the write ratio of the user's application is high enough, otherwise you should set a polling value, which must be shorter than the master's history log time.

These variables must be defined in the SPARKSEE configuration file (sparksee .cfg) or set using the SparkseeConfig class. More details on how to configure SPARKSEE can be found on the documentation site.

DEXHA activation can be checked by confirming that the Zookeeper has a node with a String containing the master's IP. # Example

Figure 1.2 is an example of a simple SPARKSEEHA installation containing:

- **HAProxy** as the **load balancer** which redirects application requests to the replicated user application instances.
- 2 Apache Tomcat servers running a **web Java user application**. In this case, those applications would be using SPARKSEEjava. Both servers run the same user application as they are replicas.
- A single-node **ZooKeeper coordinator service**.

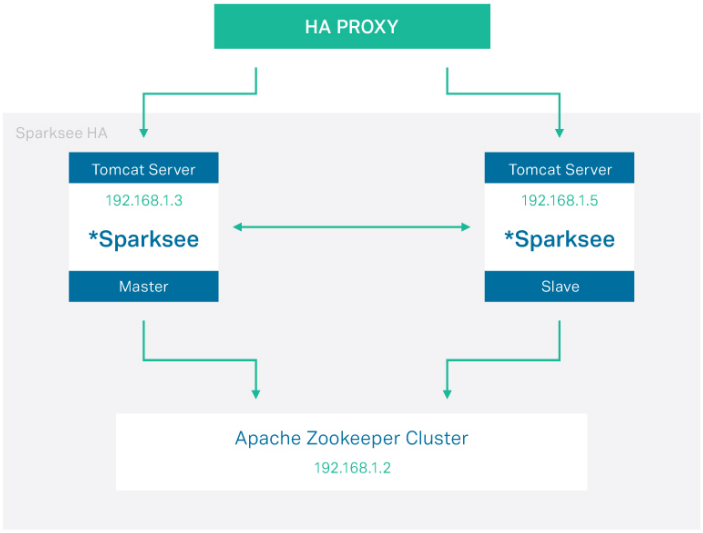


Figure 1.2: SPARKSEEHA example

HAProxy

HAProxy is a free, fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. Check their documentation site for more details about the installation and configuration of this balancer.

The configuration file for the example would look like this:

```
global
    daemon
    maxconn 500

defaults
    mode http
    timeout connect 10000ms
    timeout client 50000ms
    timeout server 50000ms

frontend http-in
    bind *:80
    default_backend sparksee

backend sparksee
    server s1 192.168.1.3:8080
    server s2 192.168.1.5:8080

listen admin
    bind *:8080
    stats enable
```

ZooKeeper

In this example, the $ZOOKEEPER_HOME/conf/zoo.cfg configuration file for the ZooKeeper server would be:

```
tickTime=2000
dataDir=$ZOOKEEPER_HOME/var
clientPort=2181
initLimit=10
syncLimit=5
```

Please note that, as it is running a single-node ZooKeeper cluster, server.x variable is not necessary.

SPARKSEEHA

The SPARKSEE configuration file for the first instance (the master) would look like this:

```
sparksee.ha=true
sparksee.ha.ip=192.168.1.3:7777
sparksee.ha.coordinators=192.168.1.2:2181
sparksee.ha.sync=600s
sparksee.ha.master.history=24H
```

And this would be the content for the file in the second instance (the slave):

```
sparksee.ha=true
sparksee.ha.ip=192.168.5.3:7777
sparksee.ha.coordinators=192.168.1.2:2181
sparksee.ha.sync=600s
sparksee.ha.master.history=24H
```

The only difference between these two files is the value of the sparksee.ha.ip variable.

As seen in the 'Architecture' chapter the role of the master is given to the first starting instance, so to make sure the instance master is that designated in the example, the order of the operations is as follows:

1. Start the master server by starting first the server with the 192.168.1.3 IP address.
2. Once the master has been started, start all the slave instances.
3. Finally, start the HAProxy.

Likewise, to shut down the system it is highly recommended that the slaves are stopped first, followed by the master.