Doradus Administration

1. Overview

This document provides configuration and operation guidelines for the Doradus database management system. It provides information on the following topics:

- Deploying Doradus in a minimal environment.
- Expanding the deployment to accommodate additional capacity.
- Security considerations and configuration.
- Cassandra and Doradus configuration options.
- Monitoring Doradus and Cassandra via JMX.

The following documents are also available:

- Doradus OLAP Database: Describes the features, data model extensions, and REST commands specific to the Doradus OLAP service.
- Doradus Spider Database: Describes the features, data model extensions, and REST commands specific to the Doradus Spider service.

1.1. Recent Changes

The following changes were made to reflect new features for the v2.3 release:

- TLS is now supported for the CQL API.
- The `doradus.yaml` configuration option `dbauthenticator` and the corresponding `SimpleAuthenticator` algorithm is no longer supported for authentication with Cassandra.
- The Doradus server now supports a "memory" appender for log details. Configuring this option in the log4j.properties file and accessing memory-based log records is described.
- Documentation for downloading, compiling, and running Doradus in various modes has been updated.
- Doradus now supports multi-tenant operation using tenant-specific keyspaces. Documentation for this feature has been added.
- New OLAP performance options cam be configured in doradus.yaml: `olap_merge_threads` and `olap_compression_threads`. These features can significantly improve OLAP shard merging performance.

2. Installing and Running Doradus

Doradus and Cassandra are both pure Java applications and can run on Windows, Linux, and OS X as console applications. On Windows, both can run as Windows services. Doradus can also be embedded within another application, meaning it runs in the same JVM. Doradus and Cassandra can be set up to run as daemon services on Linux and OS X.

2.1. Downloading Cassandra

Instructions for downloading Cassandra are available on the Apache Cassandra web site:

http://cassandra.apache.org/

Doradus works with 2.0 versions and higher. It has tested the most with the 2.0.x branch.

2.2. Downloading Doradus

Doradus is a Github project whose source code and build scripts can be found at the following link:

https://github.com/dell-oss/Doradus

There are two basic ways to download Doradus:

1) Download the entire file tree as a single zip file using this link:

   https://github.com/dell-oss/Doradus/archive/master.zip

   Unzip this file, which will create a parent folder called "Doradus-master" containing build, source, and document files. Or:

2) Clone the entire project using a Git client to your working space. For example, using a command line client:

   ```
   git clone https://github.com/dell-oss/Doradus.git
   ```

   This will create a local Git repository in a parent folder called "Doradus".

JDK 1.7 and Maven must be installed to compile Doradus. From the parent folder, enter the following Maven commands:

```
mvn clean install -DskipTests=true -Dgpg.skip=true -Dmaven.javadoc.skip=true
mvn dependency:copy-dependencies
```

The first command compiles all Doradus components: `doradus-client`, `doradus-common`, and `doradus-server`. Class and jar files are created under "target" subdirectories of each component. The second command copies dependent jar files to `target/dependency` folders of the `doradus-client` and `doradus-server` components. Therefore, you should see a directory structure similar to this:

```
./Doradus (or ./Doradus-master)
   /doradus-client
      ...
   /doradus-common
      ...
```

```
/doradus-server
    /config
    /script
    /src
    /target
        /classes
        /dependency
        ...
```

In this document, the folder `/doradus-server` is referred to as `{doradus_home}`. The next sections describe various ways to start the server.

2.3. Running Doradus Stand-Alone

Java 1.7 is required to run Doradus. It can be run as a stand-alone console application with a command executed in the `{doradus_home}` directory such as this:

```
java -cp ./config:target/classes:target/dependency/* com.dell.doradus.core.DoradusServer
```

The files `doradus.yaml` and `log4j.properties` normally reside in the folder `./config`; another location can be set with the JVM defines `-Ddoradus.config` and `-Dlog4j.configuration` respectively. Example:

```
java -Dlog4j.configuration=file:/Doradus/mylog4j.properties \
    -Ddoradus.config=file:/Doradus/mydoradus.yaml \
    ...
```

Most `doradus.yaml` parameters can be overridden with command line arguments by prepending a "-" to the parameter name. For example, `doradus.yaml` defines `restport: 1123`, which sets the REST API listening port number. To override this to 5711, add the command line argument:

```
java ... -restport 5711
```

For list value parameters, separate the values with commas and no spaces immediately after the argument name. For example:

```
-tls_cipher_suites TLS_ECDH_RSA_WITH_AES_128_CBC_SHA,TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
```

2.4. Running Doradus as a Windows Service

On Windows, Doradus can also be installed and managed as a service using the Apache Commons Daemon component. See http://commons.apache.org/proper/commons-daemon/procrun.html for more information.

2.5. Running Doradus as a Daemon

On Linux, Doradus can be executed as a daemon that is launched at system startup. Example scripts are provided in the doradus-server's scripts folder:

```
./doradus-server
    /scripts
        doradusserver
        /init.d
            doradusserver
```

The `doradusserver` script under `/init.d` can be placed in the system's `/etc/init.d` folder. See the comments in the header on modifying the file and setting permissions. The `doradusserver` file under `/scripts` is an example run file for the Doradus server. Modify it to pass parameters and JVM arguments that should be used when Doradus is started by the `init.d` script.

Doradus can also be run as a daemon on OS X using the recommended `launchd` manager. See the documentation for creating launch daemons here:

> https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html

2.6. Embedding Doradus In Another Application

Some applications may want to embed Doradus in the same JVM process. One way to do this is to call the static `main` method, passing a `String[]` that provides runtime arguments. Example:

```
import com.dell.doradus.core.DoradusServer;
...
String[] args = new String[] {"-restport", "5711"};
DoradusServer.main(args);
```

This example overrides the `doradus.yaml` parameter `restport`, setting it to `5711`. When `main()` is called, Doradus starts all internal services and the `storage_services` configured in `doradus.yaml`. However, `main()` does not return until the process receives a shutdown signal (e.g., Ctrl-C or `System.exit()` is called).

Alternatively, Doradus can be started with the method `startEmbedded()`, which returns as soon as all internal services are initialized and started. This method accepts the same `args` parameter as `main()` plus a second `String[]` that allows the selective initialization of services. Example:

```
String[] args = new String[] {"-restport", "5711"};
String[] services = new String[] {OLAPService.class.getName(), RESTService.class.getName()};
DoradusServer.startEmbedded(args, services);
```

This example also overrides the `restport` parameter, and it starts Doradus with the OLAP storage service and the REST service. Other optional services, such as the Task Manager service, are not initialized. Regardless of the services requested, Doradus always initializes required internal services such as the DB service (persistence layer) and schema service. See the doradus.yaml file for the list of optional services that may be requested through the `services` parameter.

The full package name of each service must be passed in the `services` parameter. At least one storage service must be initialized otherwise `startEmbedded()` will throw a `RuntimeException`. If multiple storage services are provided, the first one becomes the default for new applications created via the same Doradus instance that do not explicitly declare a storage service.

Note that multiple Doradus instances can be launched for the same Cassandra cluster with different service sets. For example, a direct-load application could embed Doradus, initializing only the storage service that it requires, while another stand-alone instance of Doradus can be executed with full services.

When Doradus is started with the `startEmbedded()` method, it returns as soon as all requested services are initialized and running. Doradus can be gracefully shutdown by calling the `shutDown` method. Example:

```
DoradusServer.shutDown();      // gracefully shutdown and return
```

Alternatively, Doradus can be gracefully shutdown and terminate the JVM process by calling `stopServer`. Example:

```
DordusServer.stopServer(null); // gracefully shutdown and call System.exit()
```

The parameter passed to `stopServer()` is a `String[]`, but it is ignored.

3. Deployment Guidelines

This section provides recommendations for deploying and configuring Doradus to meet various operational objectives. We start by describing the basic components of the Doradus architecture.

3.1. Basic Architecture

The basic components used by Doradus are illustrated below:

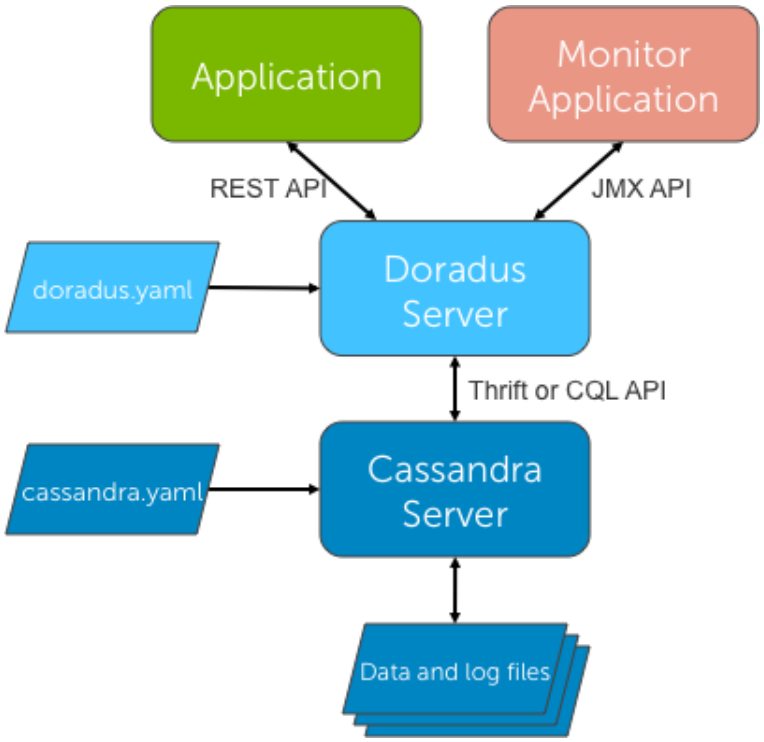

Figure 1 - Basic Doradus Components

These components are:

- **Application**: This is a Doradus client application, which performs schema commands, updates objects, and submits queries using the Doradus **REST API**.
- **Monitor Application**: Doradus provides the **JMX API** to monitor resource usage and to perform certain administrative functions such as requesting backups. A JMX application such as JConsole can be used to invoke JMX functions.
- **Doradus Server**: This is the core Doradus component, which processes REST and JMX requests. It communicates with Cassandra using either the **Thrift API** or **CQL API**. The `doradus.yaml` file is the primary Doradus configuration file.
- **Cassandra Server**: This is the core Cassandra component, which provides persistence, replication, elasticity, and other database services. Cassandra's primary configuration comes from the `cassandra.yaml` file. Cassandra stores data in various data and log files.

3.2. Minimal Configuration

Doradus uses the Cassandra NoSQL database engine for persistence, scalability, and availability. Compared to traditional relational database systems, which scale *vertically*, NoSQL databases such as Cassandra scale *horizontally*. This means that they do not require high-end servers, SAN storage, RAID configurations, and fiber channel but instead utilize a cluster of low cost, "shared nothing" servers.

Each server within a cluster is referred to a *node*. Each node adds processing capabilities to the cluster via local processes, and it adds storage capacity via locally-attached disk. For smaller deployments and testing environments, a single node can be used that runs one instance each of the Doradus Server and Cassandra Server. For a minimal level of protection, the data and log files should reside on redundant disks (e.g., RAID level 1). This is depicted below:

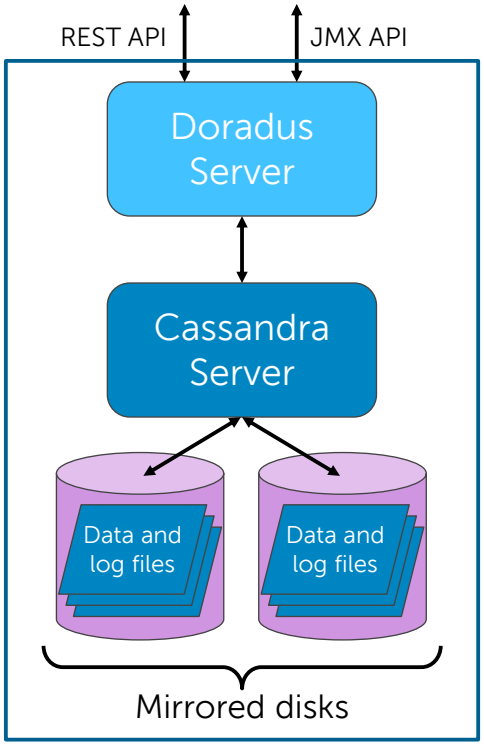

Figure 2 – Single Node Configuration

One or more application processes are assumed to reside on separate, network-near servers. In many cases, the application server process can also reside on the same node.

3.3. Two Node Configuration

To provide failure resiliency in a production deployment, at least two nodes should be used. Each node requires a Cassandra Server instance and local disk, and Cassandra must be configured with a *replication factor* of 2 (RF=2). In this configuration, there is no need for mirrored disks since each node contains a complete copy of all data. At least one node must run a Doradus Server instance. This configuration is illustrated below:

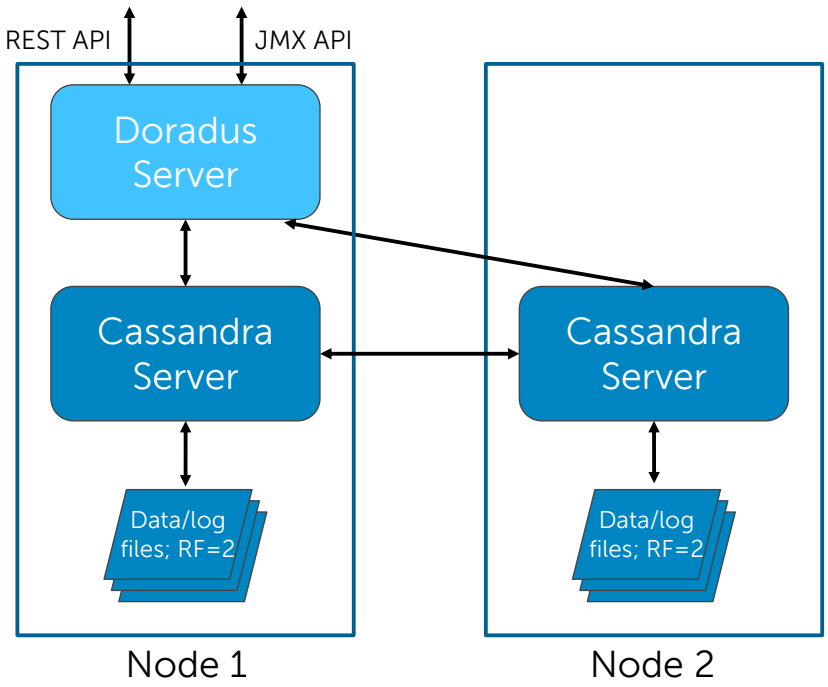

Figure 3 – Basic 2 Node Cluster: RF=2

As shown, the single Doradus Server instance receives all REST commands and JMX requests. It is configured to connect to both Cassandra instances by defining the `doradus.yaml` parameter `dbhost` as a list of both addresses. Subsequently, Doradus distributes requests to both nodes, providing load balancing. The Cassandra nodes are configured with replication factor 2 (RF=2) so that each holds an identical set of data. Should one Cassandra server instance fail, all services can be supported by the surviving node.

Multiple Doradus instances can be used in the same cluster. A typical configuration is to deploy a Doradus instance on each node but configure it to use all Cassandra instances. In a 2-node configuration, this provides full redundancy and protection against any single process or machine failure. A fully redundant 2-node configuration is show below:

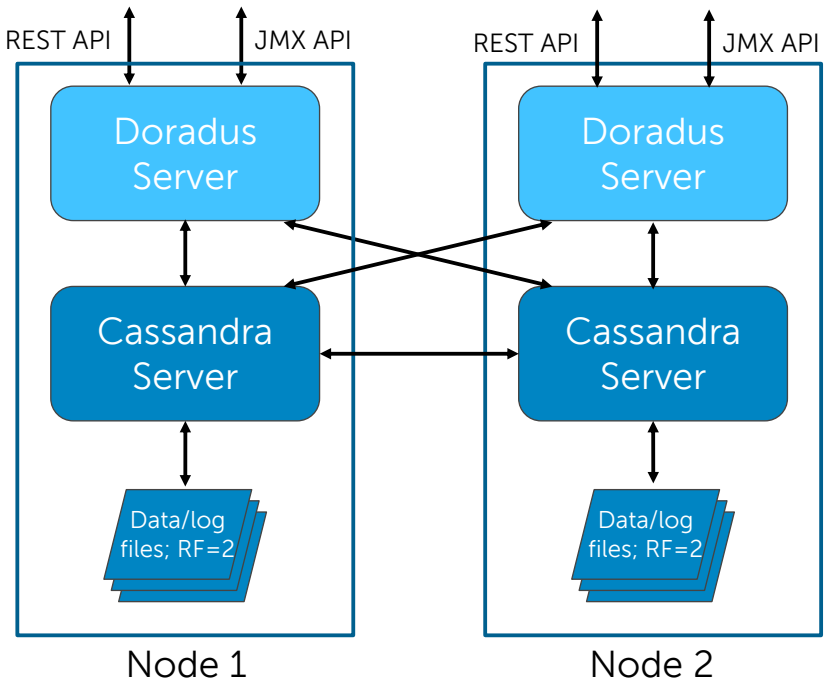

Figure 4 – Fully Redundant 2 Node Cluster: RF=2

When multiple Doradus instances are used in the same cluster, they are *peers*: any request can be sent to any instance. If a node fails, applications can redirect requests to any available instance. Doradus instances also communicate with each other to distribute background worker tasks and coordinate schema changes.

3.4. Expanding the Cluster

Additional nodes can be added to a cluster to increase processing and storage capacity. With a 3-node cluster, the recommended configuration is to use RF=2. Cassandra will distribute the data evenly within the cluster using a random partition scheme. An example 3-node cluster is illustrated below:

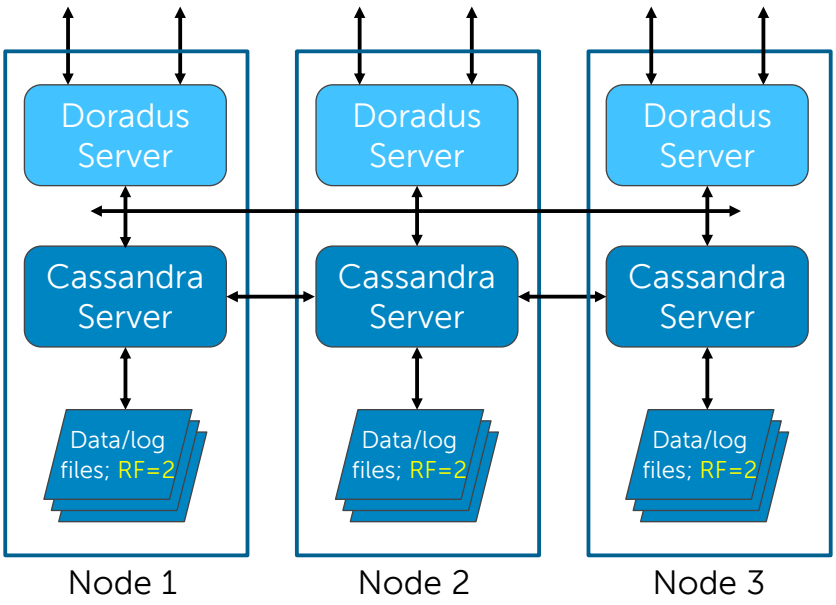

Figure 5 – Expanded Cluster: 3 Nodes, RF=2

In this configuration, every record is stored on a primary node according to its key, but RF=2 causes the record to be replicated to one other node. Hence, each node will have roughly 2/3's of the database's total records. Full database services are available if any single process or node fails. Each Doradus instance is configured to distribute requests among all Cassandra instances.

Going further up the scalability ladder, below is an example of a 5-node configuration with RF=3:

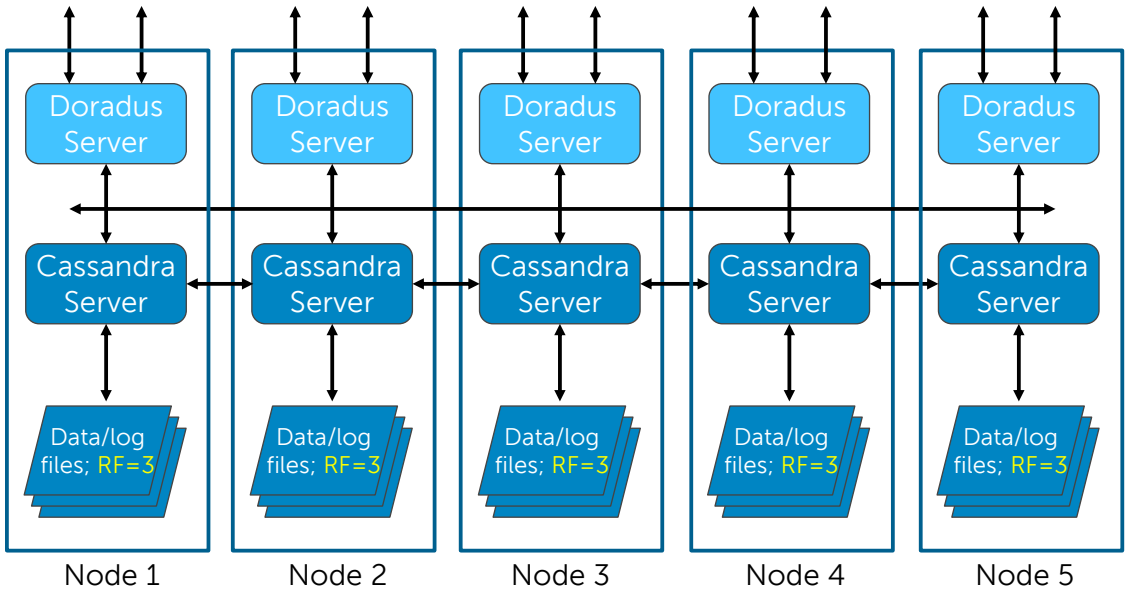

Figure 6 – Expanded Cluster: 5 Nodes, RF=3

In this example, RF=3 means that every record will be stored on a primary node and then replicated to two other nodes. This means each node has roughly 3/5's of all data. Up to 2 nodes can fail and all database services will remain available. This provides increased scalability and fault tolerance.

3.5. Multi-Data Center Deployments

Larger deployment topologies are also possible. Cassandra clusters can scale to hundreds of nodes. Each Cassandra node can also be configured with a specific *rack* and *data center* assignment. A rack is usually network-near to other racks in the same data center but independently powered. Data centers are geographically dispersed. With rack and data center awareness, Cassandra can use a replication policy that ensures maximum availability should a node, rack, or entire data center fail. An example of a multi-rack/multi-data center deployment is shown below:

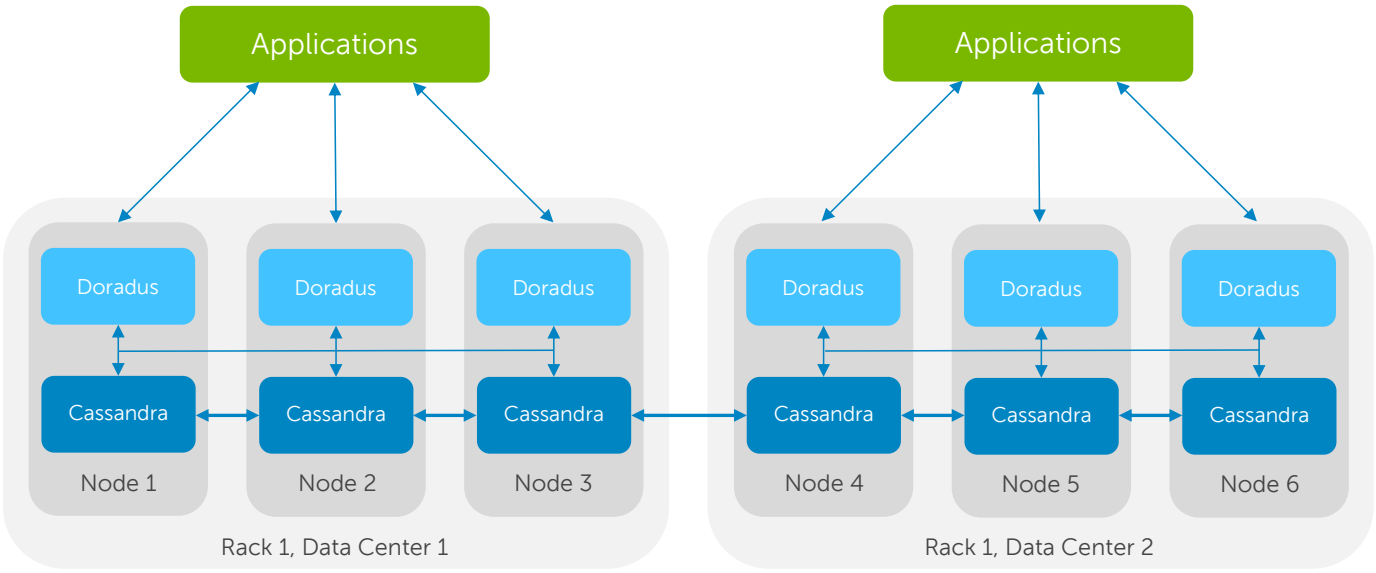

Figure 7 – Multi-Data Center Deployment

Advanced configurations such as these are beyond the scope of this document, but information about Cassandra configuration is publicly available.

3.6. Hardware Recommendations

Server class hardware is recommended for Doradus cluster nodes, though "commodity x86" based servers are sufficient, as opposed to high-end servers. The recommended hardware for each node is summarized below:

- Dual quad core Xeon processors with HT (8 cores, 16 virtual CPUs)
- 32GB of memory
- 2 x 1Gbit NICs
- 4 x hard disks (or more)

Software configuration for effectively using CPUs is described later. Two NICs are recommended, configured as follows:

- Internode traffic: Each node should have one NIC dedicated for traffic to other nodes. Cassandra will use this connection for replication and coordination. This NIC should be given a static IP address since each node will be configured to know the IP address of other nodes.

- External traffic: The other NIC should be dedicated for outbound traffic with applications.

A minimum of 4 hard disks are recommended, used as follows:

- Disk 1: Operating system and software (i.e., C: drive)
- Disk 2: Cassandra commit log
- Disks 3 and 4: Cassandra data files (2)

The software and commit log disks do not have to be large. The size and number of disks used for Cassandra data files depend on the volume of data expected. A minimum of two disks is recommended to allow parallel I/Os and improved bandwidth. When more data capacity is required for a node, added disks should be allocated to additional data files. Because resiliency is provided by internode replication, configuring disks with RAID is unnecessary.

Virtualized hosts are possible but some Cassandra experts suggest that virtualization adds a 5-15% performance penalty.

3.7. Security Considerations

This section describes security considerations that should be observed when Doradus is used to store sensitive data. Details on configuring specific Doradus and Cassandra options, such as enabling TLS for the Doradus REST protocol, are described later in sections **Doradus Security Options** and **Cassandra Security Options**.

Within the Doradus architecture, application data is stored on disk and transferred across a network. Additionally, certain configuration files contain options such as passwords and must thereby be treated as sensitive. The following diagram illustrates the files and protocols (APIs) that have potential security considerations:

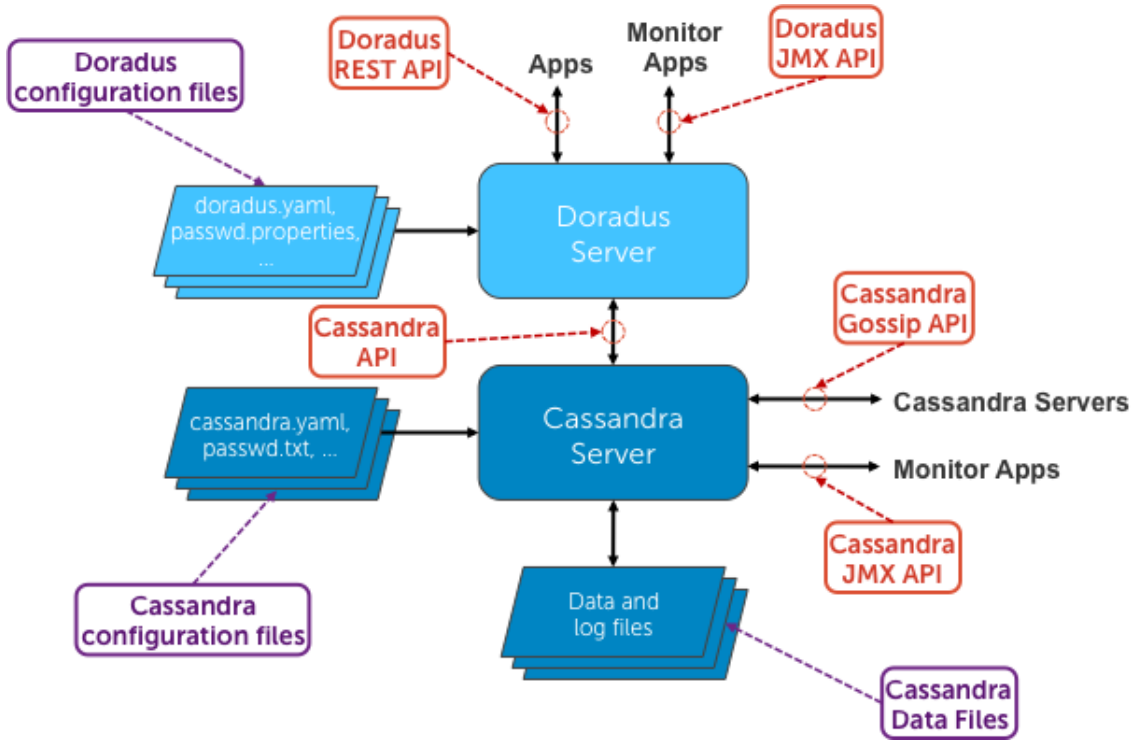

Figure 8 - Files and Protocols with Security Considerations

3.7.1. Protocols

The protocols used by Doradus and Cassandra and their security considerations are summarized below:

- Doradus REST API: This is the primary protocol used by applications to update and query objects. Doradus does not provide application-level security features: an application that can successfully connect to Doradus can update and access all objects. However, Doradus allows the REST API to be configured to use TLS (SSL), and it can restrict connections to those that provide a specific client-side certificate.

- Doradus JMX API: Doradus uses the standard Java Management Extensions (JMX) protocol for monitoring and to perform operational functions such as backup and recovery. Though application data is not transferred over the JMX API, access to it can be restricted to prevent unintended operational functions. The JMX API can be secured with authentication and/or TLS.

- Cassandra API: There are two application protocols for Cassandra: Thrift and CQL. Doradus can use either protocol: see the section Configuration Parameters in doradus.yaml. Both protocols support TLS for encryption.

- Cassandra JMX API: Cassandra also uses the JMX protocol for monitoring and to perform certain operational functions. JMX can be configured to require authorization and/or to encrypt data using TLS.

- Cassandra Gossip API: This is an inter-node communication protocol used by Cassandra to replicate data, coordinate schema changes, and perform other activities. The protocol can be configured to use TLS for encryption, however, because of the high-volume nature of this protocol, encryption is not recommended except for cross-data center communication.

3.7.2. Configuration and Data Files

The configuration and data files used by Doradus and Cassandra and their security considerations are summarized below:

- Doradus configuration files: The primary Doradus configuration file is `doradus.yaml`. This file and the `log4j.properties` file are stored in the folder `{doradus_home}/config`. These files should be considered sensitive and secured from unauthorized access.

- Cassandra configuration files: The primary Cassandra configuration file is `cassandra.yaml`. This file and other secondary configuration files are stored in the folder `{cassandra_home}/conf`. These files should be considered sensitive and secured from unauthorized access.

- Cassandra data files: All application data is stored in Cassandra data files, which reside in various folders as described in the section Setting Cassandra Data File Locations. The data in these files is unencrypted and should be secured from unauthorized access.

3.7.3. Best Practices

The recommendation best practices for security Doradus and Cassandra protocols and files are summarized below:

- **Cluster subset**: Deploy Doradus/Cassandra nodes on a subnet that is restricted from outside access. If applications reside outside of the subset, enable routing rules that restrict access to the Doradus REST API port.

- **Doradus REST API**: Secure the Doradus REST API by configuring it to use TLS. Create a client certificate that is used to restrict access to authorized applications as describe in the section **Securing the Doradus REST API**.

- **Doradus JMX API**: Use basic user ID/password authentication as described in section **Securing the Doradus JMX API**.

- **Cassandra API**: Since Doradus is intended to run on the same machine or subset as Cassandra, the Cassandra API (CQL or Thrift) does not normally need to be secured. However, TLS can be used as described in section **Securing the Cassandra API.**

- **Cassandra JMX API**: Use basic user ID/password authentication as described in section **Securing the Cassandra JMX Protocol**.

- **Doradus and Cassandra configuration files**: Secure the folders in which Doradus and Cassandra are installed, including their bin and conf or config folders.

- **Cassandra data files**: Secure the Cassandra data file folders with permissions that restrict access to the user ID under which the Cassandra process executes. For stronger security, encrypt the data within the file system, e.g., by using the Encrypted File System (EFS) on Windows.

4. Multi-Tenant Configuration

Doradus can be configured to support multiple *tenants*, which can be thought of as independent customers sharing the same cluster. Each tenant defines its own applications, whose data is physically separated from other tenants. User applications access specific tenants with tenant-specific credentials, which are implemented as Cassandra users. Hence, tenant security is at both the Doradus and Cassandra levels.

The procedures for configuring and managing multi-tenant features are described below.

4.1. Tenant Mapping to Cassandra

Each tenant is mapped to a Cassandra keyspace, meaning a keyspace is created using the tenant name. All data and metadata are stored in ColumnFamilies within the keyspace. Three ColumnFamilies are shared by all applications owned by the tenant:

- `Applications`: This ColumnFamily holds metadata for all applications owned by the tenant including schemas and, for non-default tenants, tenant options.
- `Tasks`: This ColumnFamily holds status and synchronization records used by the Doradus Task Manager service to perform background tasks such as data aging.
- `OLAP`: This ColumnFamily holds data for all Doradus OLAP applications, if any, owned by the tenant.

If the tenant owns any Doradus Spider applications, data is stored in application-specific ColumnFamilies. Two kinds of ColumnFamilies are currently used for Spider applications:

- *`application_table`*: This ColumnFamily holds object data for a specific Spider application and table.
- *`application_table`*`_terms`: This ColumnFamily holds indexing data for a specific Spider application and table.

Each tenant possesses one or more user IDs that can be used to access applications. Each user ID is mapped to a Cassandra user, prefixed with the tenant name. For example, if the tenant named `HelloKitty` owns a user called `Katniss`, the corresponding Cassandra user ID is `HelloKitty_Katniss`. This allows different tenants to independently define the same user ID. In Doradus REST commands, the tenant user ID (not the Cassandra user ID) and password are provided using basic auth.

Each tenant can also define the replication factor (RF) to use for its applications. This value cannot exceed the number of underlying Cassandra nodes. RF values greater than 1 ensure that data is replicated to 2 or more nodes, thus providing high availability in case of node failure.

4.2. Single-Tenant Operation

By default, Doradus operates in *single-tenant* mode, which means all applications are created in a single keyspace with a default name. Single-tenant mode is affected primarily by two doradus.yaml file options:

```
multitenant_mode: false
keyspace: 'Doradus'
```

When `multitenant_mode` is `false`, all applications are stored in the *default* keyspace defined by the `keyspace` option.

Single-tenant mode does not require security to be enforced within Cassandra. However, you can configure Cassandra to enforce user/password authentication by setting the following options in each Cassandra node's cassandra.yaml:

```
authenticator: PasswordAuthenticator
authorizer: CassandraAuthorizer
```

When Cassandra first starts with these options, it creates a default *super user* account with the user ID and password `cassandra`/`cassandra`. You should change the super user ID and/or password to something more secure. Doradus must use a super user account for access, hence set the following doradus.yaml options to a valid super user account:

```
dbuser: cassandra
dbpassword: cassandra
```

You can also pass these options into Doradus as runtime as arguments "`-dbuser xxx -dbpassword yyy`".

4.3. Multi-Tenant Operation

Multi-tenant operation is enabled by setting `multitenant_mode` to `true`. Multi-tenant mode requires the use of the CQL API, hence `use_cql` must also be set to `true`, and `dbport` should be set to Cassandra's CQL port. Finally, multi-tenant mode requires Cassandra to enforce user ID/password authentication, and Doradus must be configured to use a super user. Therefore, enabling multi-tenant mode requires setting all of the following doradus.yaml options:

```
multitenant_mode: true
use_cql: true
dbport: 9042
dbuser: cassandra
dbpassword: cassandra
```

As described in the previous section, you can pass `dbhost` and `dbpassword` as runtime arguments to prevent storing them in the doradus.yaml file.

When multi-tenant mode is enabled, all application-specific REST commands that do not identify a specific tenant are directed to the default tenant. Such commands do not require tenant credentials, hence the default tenant can be considered a "public" tenant. You can disable the default tenant by setting the following doradus.yaml option:

```
disable_default_keyspace: true
```

When this option is set, all application-specific REST commands must be directed to a specific tenant and provide valid credentials for that tenant.

When a new tenant is defined, Doradus creates the corresponding Cassandra keyspace using the following doradus.yaml file options as a template:

```
ks_defaults:
    - strategy_class: SimpleStrategy
    - strategy_options:
        - replication_factor: "1"
    - durable_writes: true
```

This options can be changed to use different defaults for new keyspaces. In a multi-node cluster, replication_factor should be increased to ensure data availability in case of a node failure. Note a tenant can be creating while overriding some of these default keyspace options.

4.4. Managing Tenants

The REST commands to create, modify, and delete tenants are considered privileged operations, hence they must be accompanied with valid super user credentials. REST commands use *basic authorization* (basic auth), which means credentials are passed using the `Authorization` header:

```
Authorization: Basic xxx
```

Where xxx is the user ID and password, separated by a colon, and then base64 encoded. For example, if the super user and password are `cassandra:cassandra`, the header would look like this:

```
Authorization: Basic Y2Fzc2FuZHJhOmNhc3NhbmRyYQo=
```

4.4.1. Creating a New Tenant

A new tenant is created with the REST command:

```
POST /_tenants
```

This command must include a message body that minimally identifiers the new tenant's name. Optionally, the command can define one or more tenant users and/or tenant options. An example in XML is shown below:

```
<tenant name="HelloKitty">
   <users>
      <user name="Katniss" password="Everdeen"/>
   </users>
   <options>
      <option name="ReplicationFactor">3</option>
   </options>
</tenant>
```

In JSON:

```
{"HelloKitty": {
   "users": {
      "Katniss": {"password": "Everdeen"}
   },
   "options": {
      "ReplicationFactor": "3"
   }
}}
```

Only the new tenant name (`HelloKitty`) is required. Optionally, one or more tenant user/password can be defined (e.g., `Katniss`/`Everdeen`) with a `users` group. A tenant `options` group is also allowed, though the only option currently supported is `ReplicationFactor`. When set, this option overrides the `replication_factor` defined with the `ks_defaults` parameter (in doradus.yaml).

When a `POST /_tenants` command is received, if the given tenant already exists, a `409 Conflict` error response is returned. (This means tenant creation is not idempotent—a POST of the same request will produce an error.) If the tenant does not already exist, the following occurs:

- A new keyspace is created using the tenant name. If the option `ReplicationFactor` is specified, it is used for the new keyspace. All other defaults come from the doradus.yaml parameter `ks_defaults`.
- If no tenant users are specified, a new random user ID and password are chosen for the tenant.
- CQL commands are issued to create all new tenant user IDs/passwords and authorize them for full access to the new keyspace. The Cassandra user is named `{tenant}_{user}` so that user names are unique across tenants. The Cassandra user can be used for direct access to Cassandra.
- The keyspace is initialized by creating the `Applications` and `Tasks` ColumnFamilies.
- The tenant's definition is written to the `Applications` CF so it can be recovered if needed.

The POST command returns the new tenant's full definition, including name, user IDs/passwords, and options using the same document format shown above.

4.4.2. List All Tenants

This command lists all existing tenants:

```
GET /_tenants
```

This command returns each tenant's name and the list of applications owned by each tenant. An example response in XML is shown below:

```
<tenants>
    <tenant name="Doradus">
        <applications>
            <value>SmokeTest</value>
            <value>App_Default</value>
        </applications>
    </tenant>
    <tenant name="Bibliotecha">
        <applications>
            <value>App_Biblio</value>
        </applications>
    </tenant>
</tenants>
```

As with all REST commands, a JSON response can be requested by adding `?format=json` to the URI. Example:

```
GET /_tenants?format=json
```

Which returns:

```
{"tenants": {
    "Doradus": {
        "applications": [
          "SmokeTest",
          "App_Default"
        ]
    },
    "Bibliotecha": {
        "applications": [
          "App_Biblio"
```

```
        ]
    }
}}
```

4.4.3. List Tenant

Details of a specific tenant can be listed with the following REST command:

```
GET /_tenants/{tenant}
```

Where {tenant} is the tenant name. This command returns all tenant details includes its name, the list all tenant user IDs and passwords, and any special options used to create the tenant. An example in XML is shown below:

```
<tenant name="Bibliotecha">
    <users>
        <user name="Uqyam0my7ivcm" password="r13dc42ppyu8"></user>
    </users>
</tenant>
```

4.4.4. Modify Tenant

An existing tenant can be modified via the following REST command:

```
PUT /_tenants/{tenant}
```

Where {tenant} is the tenant name. Currently, the only changed allowed are modifications to the tenant's users: New users can be added and the password of existing users can be modified. The command must include an XML or JSON document with the same format used to create a tenant. Example:

```
<tenant>
    <users>
        <user name="Katniss" password="MockingJay"/>
        <user name="Rumpleteazer" password="Mongojerrie"/>
    </users>
</tenant>
```

In this example, the password for user `Katniss` is modified and a new user `Rumpleteazer` is added.

4.4.5. Delete Tenant

An existing tenant can be deleted via the following REST command:

```
DELETE /_tenants/{tenant}
```

Where {tenant} is the tenant name. No message body is provided in the command. The tenant's keyspace, all of its applications, and of its data are deleted. (Note that Cassandra creates a snapshot of a CF when it is deleted, so the data is still recoverable for a while.)

4.5. Application Commands in Multi-Tenant Mode

Application REST commands are those that define or modify application schemas, update data, query data, or perform other application-specific commands supported by the managing storage service. When Doradus is operating in multi-tenant mode, all such commands must identify the target tenant by

appending `?tenant={tenant}` to the URI. For example, the following commands are directed to the tenant named `HelloKitty`:

Examples:

```
POST /_applications?tenant=HelloKitty        // create a new application
POST /foo/bar?tenant=HelloKitty              // add data to application foo
GET /_tasks?tenant=HelloKitty                // list tasks for all applications
GET /_olapp?tenant=HelloKitty                // display the OLAP browser
```

If no `?tenant` parameter is provided, the command is directed to the default tenant and no credentials are required. (Note that access to the default tenant may be disabled if the option `disable_default_keyspace` is set to `true`.) When a REST command uses other URI query parameters, the `tenant` parameter can be given anywhere in the query string. URI query parameters are separated by the ampersand (&). The following commands are equivalent, sending a query to the application `foo` in the `HelloKitty` tenant:

```
GET /foo/bar/_query?q=*&tenant=HelloKitty
GET /foo/bar/_query?tenant=HelloKitty&q=*
```

Tenant-specific application commands must be accompanied with valid credentials for that tenant. Credentials are passed using a *basic authorization* (basic auth) header `Authorization`. Example:

```
Authorization: Basic xxx
```

Where `xxx` is the tenant user ID and password, separated by a colon, and base64-encoded. For example, if the user ID and password are `Katniss:Everdeen`, the header would look like this:

```
Authorization: Basic S2F0bmlzczpFdmVyZGVlbgo=
```

When Doradus receives this header, the base64 value is decoded and validated against the given tenant. Note that curl supports basic authentication by adding the "-u" parameter. Example:

```
curl -u Katniss:Everdeen http://localhost:1123/HelloKitty/...
```

If the tenant user ID or password is incorrect for the identified tenant, the REST command returns a `401 Unauthorized` response.

Note that super user credentials can be used to access any tenant, hence they must be kept secured.

4.6. System Commands in Multi-Tenant Mode

REST commands that are not application-specific are called *system commands* and are not directed to a specific tenant and hence do not use the `?tenant` parameter. This includes the following URIs:

```
/_tenants/...                        // Tenant manager command
/_logs                               // Dump logs command
/_dump                               // Dump threads command
```

In multi-tenant mode, system commands must also be authenticated via basic auth using super user credentials.

5. Doradus Configuration and Operation

This section provides recommendations for runtime and configuration options for the Doradus server.

5.1. Runtime Memory

The default JVM memory is probably for a Doradus Server doing anything important. Also, tests have shown that *growing* a Java process's memory can be very disruptive, therefore, we recommend setting both the minimum and maximum memory parameters to the same value. 4GB is a good value to start with. Example:

```
java -Xms4G -Xmx4G -cp "../lib/*:../config/*" com.dell.doradus.core.DoradusServer
```

5.2. Doradus Configuration Files

The following are the primary Doradus configuration files within the `{doradus_home}/config` folder:

- **`doradus.yaml`**: This file provides global configuration parameters. Typically only a few parameters merit modification from their defaults.
- **`log4j.properties`**: This file controls logging features of the Doradus instance.

5.2.1. Setting Doradus Logging Options

Doradus logs messages about its operation using the log4j logging facility. Logging options are defined in the file `log4j-server.properties`. The high-level logging options are shown below:

```
log4j.rootLogger=DEBUG, console, file, memory
log4j.logger.org.eclipse.jetty=INFO
log4j.appender.console.Threshold=INFO
log4j.appender.file.Threshold=INFO
log4j.appender.memory.Threshold=DEBUG
```

This sets the global logging level to DEBUG and defines multiple "appenders": console (stdout), a disk file, and memory. Although DEBUG-level log records are generated, only INFO-level entries (and above) are sent to the console and file appenders. DEBUG entries are only collected by the memory appender. This minimizes the output to the console- and file-based logs, but it allows the most recent DEBUG records to be retrieved via the REST command: `GET /_logs`.

If Doradus is run as a service, the output to console is unnecessary, so this parameter should be removed. Additionally, the lines that begin with `log4j.appender.console` should be deleted or commented-out.

Because the memory appender only retains the most recent log entries, in diagnostic situations, you might want to capture DEBUG entries in the file log as well. Note that this causes log files to grow more quickly. DEBUG entries can be captured in the file appender by changing the following line:

```
log4j.appender.console.Threshold=DEBUG
```

The `log4j-server.properties` file also defines the following option:

```
log4j.appender.file.File=doradus.log
```

This option sends the file log appender to a log file called `doradus.log` in the Doradus server's runtime directory. If you wish log files to be stored with another name and/or in another location, modify this option with a new relative or absolute file name.

Two other options to consider:

```
log4j.appender.file.maxBackupIndex=50
log4j.appender.file.maxFileSize=20MB
```

These options cause the log file to grow up to 20MB, at which point it is renamed with a numeric extension (.1, .2, etc.) and a new file is started. Up to 50 files are retained (up to 1GB total log data), at which point the oldest file is removed. Modify these options to increase or decrease the number and size of log files stored.

5.2.2. Doradus Security Options

Doradus does not store any application data; all data is stored by the Cassandra database service. Securing Doradus from unauthorized access requires securing its configuration files at the network protocols.

5.2.2.1. Securing the Doradus REST API

By default, the Doradus REST API uses unencrypted HTTP. Because Doradus provides no application-level security, any process that connect to the Doradus REST port is allowed to perform all schema, update, and query commands. The REST API can be secured by enabling TLS (SSL), which encrypts all traffic and uses mutual authentication to restrict access to specific clients. Optionally, client authentication can be enabled to restrict connections to only those whose certificates have been registered at the server. The process for securing the REST port with TLS is defined below:

1) Enable TLS by setting the `tls` parameter in the `doradus.yaml` file to true. Example:

   ```
   tls: true
   ```

2) Create a certificate for use by the Doradus server and store it in a keystore file. You can use the `keytool` utility included with the JRE. An overview of the process to create a self-signed certificate is outline here:

   http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CreateKeystore

3) Set the `keystore` parameter in the `doradus.yaml` file to the location of the **keystore** file, and set the `keystorepassword` parameter to the file's password. Example:

   ```
   keystore: config/keystore
   keystorepassword: mykspassword
   ```

4) If client authentication will be used, create a certificate for each client application and import them into a **truststore** file. (See the same article referenced above.) Set the `truststore` parameter in the `doradus.yaml` file to the location of the keystore file, and set the `truststorepassword` parameter to the file's password. Example:

   ```
   truststore: config/truststore
   truststorepassword: mytspassword
   ```

5) To require client authentication, set the `clientauthentication` parameter in the `doradus.yaml` file. This requires REST API connections to use mutual authentication. Example:

```
clientauthentication: true
```

6) The cipher algorithms allowed by the REST API when TLS is enabled is controlled via the `tls_cipher_suites` parameter. The default list includes the algorithms recommended for FIPS compliance. The actual algorithms allowed by REST API is a subset of the listed algorithms and those actually available to the JVM in which Doradus is running. The cipher algorithm list can be tailored, for example, to only allow 256-bit symmetrical encryption. Example:

```
tls_cipher_suites:
   - TLS_DHE_DSS_WITH_AES_256_CBC_SHA
   - TLS_DHE_RSA_WITH_AES_256_CBC_SHA
   - TLS_RSA_WITH_AES_256_CBC_SHA
   - TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
   - TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
   - TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
   - TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
```

   Custom algorithms can also be used as long as installed with the JVM used to run Doradus.

With the steps above, Doradus will use TLS for its REST API port, optionally requiring mutual authentication. Clients must connect to the REST port using TLS. If client authentication is enabled, they must submit a certificate that was added to the truststore. Each client must also support one of the configured cipher algorithms.

5.2.2.2. Securing the Doradus JMX API

Doradus supports the Java Management Extensions (JMX) protocol for monitoring Doradus and performing certain administrative commands. JMX can be secured with authorization and/or encryption, however, security JMX security is disabled in the `{doradus_home}/bin/doradus-server.bat` file included with Doradus. The common JMX options are defined in variables at the top of the file as shown below:

```
set JMX_PORT=9999
set JMX_SSL=false
set JMX_AUTHENTICATE=false
```

These options are summarized below:

- **`JMX_PORT`**: This variable sets the `com.sun.management.jmxremote.port` define in the JVM, which is the port number that the remote JMX clients must use. As shown, port 9999 is the default used for Doradus.

- **`JMX_SSL`**: This variable sets the `com.sun.management.jmxremote.ssl` define in the JVM. When set to `true`, it requires remote clients to use SSL (TLS) to connect to the JMX port. When SSL is enabled, additional options are available to require remote clients to have a client-side certificate.

- **`JMX_AUTHENTICATE`**: This variable sets the `com.sun.management.jmxremote.authenticate` define in the JVM. When set to true, it requires remote clients to provide a user ID and password in order to connect. Additional parameters are required to define allowed user IDs and passwords and the locations of the corresponding files.

Because JMX is a standard and external documentation is available for securing JMX, details for using SSL and/or authentication are not covered here. See for example the following:

http://docs.oracle.com/javase/6/docs/technotes/guides/management/agent.html

Though not shown above, here is another useful option: On multi-homed systems, `java.rmi.server.hostname` can be set to a specific IP address, causing JMX to bind to that address instead of the default one. For example:

```
set JAVA_OPTS=
...
 -Djava.rmi.server.hostname=10.1.82.121^
...
```

This causes the JMX port to bind to address 10.1.82.121.

5.2.2.3. Using a Secured CQL API

As described in the section **Securing the Cassandra API**, by default Cassandra's CQL and Thrift APIs are not secured. Encrypted communication can be enabled as described in that section.

5.2.3. Configuring Doradus for Clusters

Multiple instances of Cassandra and/or Doradus can be used in the same cluster. As described in the section **Deployment Guidelines**, a recommended practice is to deploy an instance of both Doradus and Cassandra on each node. Then, each Doradus instance should be configured to contact all Cassandra nodes. This provides load balancing and fault tolerance in case a Cassandra node fails.

The Cassandra instance(s) to which each Doradus instance connects is defined in the `dbhost` parameter in its `doradus.yaml` file. This parameter should be set to a comma-separated list of host names or (preferably) IP addresses of Cassandra instance hosts. For example:

```
dbhost: 10.1.82.146, 10.1.82.147, 10.1.82.148
```

This example tells the corresponding Doradus instance to use three Cassandra instances. Doradus will distribute connections across these nodes for load balancing.

Note that when Doradus is configured to use the CQL API, all Cassandra nodes will be automatically discovered and used. Therefore, the `dbhost` parameter only needs to list one of the available nodes.

5.2.4. Configuration Parameters in `doradus.yaml`

This section describes the parameters that can be set in the `doradus.yaml` file. Except as discussed in the previous sections, most of these parameters can be left at their default value. However, there are some values that you should change for clustered configurations, to enable OLAP caching, etc.

5.2.4.1. Multi-tenant Mode Parameters

See the section Multi-Tenant Configuration for details on configuring and operating Doradus single- and multi-tenant modes. Below is a reference of the parameters that affect multi-tenant operation:

```
multitenant_mode: false
```

This parameter controls whether Doradus operates in single-tenant mode (false) or multi-tenant mode (true).

```
disable_default_keyspace: false
```

When Doradus is operating in multi-tenant mode, this parameter controls whether access to the default tenant is allowed.

```
keyspace: Doradus
```

This parameter defines the name of the default tenant.

```
ks_defaults:
    - strategy_class: SimpleStrategy
    - strategy_options:
        - replication_factor: "1"
    - durable_writes: true
```

This parameter group is used when creating both the default tenant and new tenants in multi-tenant mode.

Note that when multi-tenant mode is enabled, `use_cql` must be set to `true`, and `dbuser` and `dbpassword` must be set to the credentials for a Cassandra super user account. These parameters are described later.

5.2.4.2. Default Services Parameter

Default services define the internal Doradus services that will be started when Doradus is run in stand-alone mode. (When Doradus is started in embedded mode, the default services are passed to the `DoradusServer.startEmbedded()` method.) These services are in addition to storage services, which are defined separately. `DBService` and `SchemaService` are required and always started even if not listed here. Other services are optional and can be disabled when not needed. Custom services can also be added to this list. Note that the default services parameter is only used by Doradus in stand-alone mode. When Doradus is started as an embedded application, all services (include storage services) must be passed to the `startEmbedded()` method. Each line must contain the full package name of the service class. The optional services are:

```
com.dell.doradus.mbeans.MBeanService
```

Provides monitoring and management services via custom Doradus MBeans. Can be disabled if custom JMX commands are not needed.

```
com.dell.doradus.service.rest.RESTService
```

Provides the REST API via an embedded Jetty server, which listens to the server and port defined by `restaddr` and `restport`. It only makes sense to disable this when running Doradus as an embedded service.

```
com.dell.doradus.service.taskmanager.TaskManagerService
```

Provides background task management for data aging, statistics refreshing (Spider), and other scheduled tasks. Also manages on-demand tasks such as removing obsolete data (Spider). Can be disabled if background tasks are not needed for this Doradus instance.

```
default_services:
    -com.dell.doradus.mbeans.MBeanService
    -com.dell.doradus.service.db.DBService
    -com.dell.doradus.service.rest.RESTService
    -com.dell.doradus.service.schema.SchemaService
    -com.dell.doradus.service.taskmanager.TaskManagerService
```

5.2.4.3. Storage Services Parameter

The `storage_services` parameter defines which storage services are initialized when Doradus is started as a standalone application (e.g., as a service or console app). At least one storage service must be defined. The first storage service listed becomes the default for new applications created without explicitly declaring a storage service. Example:

```
storage_services:
    - com.dell.doradus.service.spider.SpiderService
    - com.dell.doradus.service.olap.OLAPService
```

This initializes both the Spider and OLAP services, making the Spider service the default for new applications.

5.2.4.4. Cassandra ColumnFamily Parameters

The `cf_defaults` parameter is used when Doradus creates a new ColumnFamily. All values recognized by Cassandra are allowed and are passed "as is". Parameters must begin with a dash (-), and sub-parameters should be further indented. Empty values should be specified with a pair of quotes. Example:

```
cf_defaults:
    - compression_options:
        - sstable_compression: ""   # use empty string for "none"
    - gc_grace_seconds: 3600
```

After a ColumnFamily is created, Doradus does not modify its parameters to match values in `doradus.yaml`. It is safe to use Cassandra tools such as `cassandra-cli` to modify ColumnFamily parameters when needed.

See Cassandra documentation for information about ColumnFamily parameters.

5.2.4.5. Cassandra Connection Parameters

These parameters control connections to the Cassandra database using the Thrift or CQL APIs:

`dbhost: 'localhost'`

This parameter configures the host name(s) or address(es) of the Cassandra node(s) that the Doradus server connect to. Static IP addresses are preferred over host names. If a comma-separated list of hosts is provided, Doradus will connect to each one in round-robin fashion, allowing load balancing and failover.

`dbport: 9160`

This is the Cassandra Thrift or CQL API port number. Change this value if Cassandra's port is using something other than 9160. (Cassandra should use the same port on all nodes.)

`jmxport: 7199`

Change this value if Cassandra's JMX port is using something other than 7199. (Cassandra should use the same port on all nodes.)

`db_timeout_millis: 60000`

This is the socket-level Cassandra database connection timeout in milliseconds. It is applicable to both the Thrift and CQL APIs. This time is used both when connecting to Cassandra and when performing read/write operations. If a connect, read, or write request is not received within this time, the connection is closed and a retry is initiated. Note, however, that Cassandra has its own RPC timeout value, which defaults to 10 seconds. This means that read and write operations will typically based on Cassandra's value and not this one.

`db_connect_retry_wait_millis: 30000`

This is the Cassandra database (Thrift) initial connection retry wait in milliseconds. It is applicable to both the Thrift and CQL APIs. If an initial connection to Cassandra times-out (see `db_timeout_millis` above), Doradus waits this amount of time before trying again. Doradus keeps retrying indefinitely under the assumption that Cassandra may take a while to start.

`max_commit_attempts: 10`

This is the maximum number of times Doradus will attempt to perform a Cassandra update operation before aborting the operation. (See also `retry_wait_millis`).

`max_read_attempts: 3`

This is the maximum number of times Doradus will attempt to perform a read operation before aborting the operation. (See also `retry_wait_millis`).

`max_reconnect_attempts: 3`

This is the maximum number of times Doradus will attempt to reconnect to Cassandra after a failure. A reconnect is performed if a read or update call fails. (See also `retry_wait_millis`).

`retry_wait_millis: 5000`

This is the time in milliseconds that Doradus waits after a failed update, read, or reconnect request call to Cassandra. This time is multiplied by the attempt number, which means Doradus waits a little longer after each successive failed call. (See also `max_commit_attempts`, `max_read_attempts`, and `max_reconnect_attempts`.)

`use_cql: false`

By default, Doradus uses the Thrift API to communicate with Cassandra. When this option is set to `true`, it uses the CQL API instead. CQL must be enabled in each Cassandra instance that Doradus connects via the `cassandra.yaml` file option `start_native_transport: true`. Additionally, the `doradus.yaml` file option `dbport` must be set to the CQL API's port (the default is `9042`).

Note: A Doradus database created with the v2.1 or prior release must continue to use the Thrift API. New databases created with the v2.2 release can use either Thrift or CQL. Tests show that the Thrift API is a little faster for bulk data loading by about 3%. But over time, CQL may be faster due to the advantages of prepared statements. CQL is the recommended Cassandra API, and over time the Thrift API may be phased-out.

`dbtls: false`

Set to true to use TLS/SSL for connections to Cassandra. This option works for both CQL and Thrift. When `dbtls` is enabled, the options `keystore` and `keystorepassword` are used for certificate information.

```
dbtls_cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA]
```

Set to a list of one or more cipher suites to be used with SSL/TLS to Cassandra. This option is only meaningful if `dbtls` is set to `true`. Cassandra must be configured to allow at least one of the same cipher algorithms in order to establish a successful connection. The list can be comma-separated on a single line within square brackets (e.g., [X, Y, Z]), or each cipher suite can be listed on its own line, indented and prefixed with a "-". The cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` is generally always allowed by Cassandra.

```
dbuser: cassandra
dbpassword: cassandra
```

If authentication/authorization is configured for Cassandra, these parameters must be set to a super user that Doradus can use. When Cassandra's authenticator is set to `PasswordAuthorizer`, it initially creates the super id/password `cassandra`/`cassandra`. A better password and/or a different super user should be created for Doradus. Remember that these values can be passed-in dynamically as run-time arguments.

5.2.4.6. Doradus REST API Parameters

These parameters configure the Doradus REST API.

```
restaddr: 0.0.0.0
```

By default, the Doradus REST API binds to address 0.0.0.0, which means it accepts connections directed to any of the host's IP addresses (including "localhost"). Change this value if you want the REST API to accept connections on a specific IP address instead.

```
restport: 1123
```

This is the REST port that Doradus listens to. Change this to accept REST API connections on a different port.

```
maxconns: 200
```

This is parameter is used to configure the maximum threads that the internal Jetty server allows. More than this number of connections may be received, but this value controls the maximum number of parallel REST commands that may be in progress at one time.

```
tls: false
tls_cipher_suites:
   - TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
   ...
clientauthentication: false
keystore: config/keystore
keystorepassword: changeit
truststore: config/truststore
```

```
truststorepassword: password
```

As described in the section 5.2.2 Doradus Security Options, these parameters should be set to configure TLS/SSL for the Doradus REST API and to restrict connections to authenticated clients.

```
max_request_size: 52428800
```

This parameter determines the maximum size of an input requested that is accepted. Requests larger than this are rejected and the socket is closed. The default is 50MB.

5.2.4.7. Doradus Query Parameters

These parameters configure defaults used for Doradus object and aggregate query parsing and execution.

```
search_default_page_size: 100
```

This value is the default query page size used when an object query does not specify the &s (page size) parameter.

```
aggr_separate_search: false
aggr_concurrent_threads: 0
dbesoptions_entityBuffer: 1000
dbesoptions_linkBuffer: 1000
dbesoptions_initialLinkBuffer: 10
dbesoptions_initialLinkBufferDimension: 1000
dbesoptions_initialScalarBuffer: 30
l2r_enable: true
```

These parameters are set by the Doradus team and normally should not be changed.

5.2.4.8. OLAP Parameters

These parameters control the operation of the OLAP service:

```
olap_cache_size_mb: 100
```

This parameter controls the memory size of the most-recently-used field cache. If this value is exceeded, older fields will be un-loaded from memory.

```
olap_file_cache_size_mb: 100
```

When this value is non-zero, it enables OLAP "file" caching and sets the total size, in megabytes, of the cached data. OLAP uses virtual files to hold raw scalar values such as text values. Caching this data prevents round-trips to the database for certain types of queries. This value does not affect shard caching defined by other parameters.

```
olap_query_cache_size_mb: 100
```

When this value is non-zero, it enables recent query result caching and sets the total size, in megabytes, of the cached search results. Each cached result takes 1 bit per each object in the table.

```
olap_cf_defaults:
    - compression_options:
        - sstable_compression:  "" # use empty string for "none"
    - gc_grace_seconds: 3600
```

This parameter is used to create the OLAP ColumnFamily. All parameters recognized by Cassandra are accepted and passed "as is". Parameters must be indented and begin with a "-"; sub-parameters must be further indented. The OLAP ColumnFamily is created when the server is first started for a new database. If the OLAP ColumnFamily already exists, it is not modified to match these parameters. See Cassandra documentation for details about these values. Note that Doradus automatically compresses data before storing in Cassanda, hence compression should be disabled for the OLAP ColumnFamily.

```
olap_merge_threads: 0
```

This parameter controls the number of threads used to merge data within a shard. The default value of 0 means that a single thread is used to merge all data. When this value is > 0, multiple threads are used to merge shard data in parallel. Increasing this value can significantly decrease the time needed to merge shards. However, it must be balanced with the number of processors available on the machine.

```
olap_compression_threads: 0
```

This parameter controls the number of threads used to compress data before storing in Cassandra. This parameter interacts with `olap_merge_threads` as follows:

- If both `olap_merge_threads` and `olap_compression_threads` = 0, then a single thread is used to merge and store all data for each thread.
- If `olap_merge_threads` > 0 and `olap_compression_threads` = 0, then each merge thread both merges data and compresses all segments before writing to Cassandra.
- If `olap_compression_threads` > 0, then the merge thread(s) create data segments that are then queued for compression in the specified number of asynchronous compression threads.

Setting `olap_compression_threads` > 0 can significantly decrease the time needed to merge large OLAP shards. However, the value used must be balanced with `olap_merge_threads` and the number of cores available on the system.

One parameter worth highlighting is the OLAP ColumnFamily's `gc_grace_seconds`. This value controls how long deleted rows called "tombstones" are retained with data tables (called *sstables*) before they are truly deleted during a compaction cycle. The default is 864,000 seconds or 10 days, which provides lots of time for a failed node to recover and learn about deletions it may have missed. However, the OLAP service deletes many rows when it merges a shard, and these rows consume disk space. Moreover, they consume memory because active sstables are memory-mapped (mmap-ed) by Cassandra. This can cause excessive memory usage.

A much smaller `gc_grace_seconds` value is recommended for the OLAP CF, somewhere between 3600 (a hour) and 86400 (1 day). This causes tombstones to be deleted more quickly, freeing-up disk space and reducing memory usage.

5.2.4.9. Spider Parameters

These parameters control the operation of the Spider server:

```
batch_mutation_threshold: 10000
```

Defines the maximum number of mutations a batch update will queue before flushing to the database. After the mutations are generated for each object in a batch update, Spider checks to see if this threshold has been met and flushes the current mutation batch if it has. Larger values improve performance by reducing Cassandra round trips, but they require larger API buffer sizes. The default is 10,000.

5.3. Managing Doradus via JMX

Doradus extends the Java JMX API with monitoring and administration features. JMX functions can be accessed with off-the-shelf JMX applications such as JConsole, which is bundled with the Java SDK. JMX functions are available from two `MXBean` objects:

- `com.dell.doradus.ServerMonitor`: This `MXBean` provides information about the Doradus server including its configuration, start time, and statistics about REST commands.
- `com.dell.doradus.StorageManager`: This `MXBean` provides information used by the Doradus Task Manager to conduct background tasks. It also provides operations for modifying tasks and to execute Cassandra backup and recovery tasks.

JMX functions only collect and return information when the Doradus MBean service (`com.dell.doradus.mbeans.MBeanService`) has been initialized. This is always the case when the server is started as a standalone process but optional when Doradus is embedded within another application.

5.3.1. ServerMonitor

`ServerMonitor` provides the following attributes:

Attribute Name	Description
`AllRequests`	Provides a composite set of values about REST API requests such as the total number of requests, the number of failed requests, and the reason of the last request failure.
`ConnectionsCount`	The current number of REST API connections.
`DatabaseLink`	Indicates if the Cassandra database being accessed is local (1), remote (2), or not yet known (0).
`RecentRequests`	A composite value set similar to `AllRequests` but for REST API requests made since the last time `RecentRequests` was accessed.
`ReleaseVersion`	The Doradus Server release version.
`ServerConfig`	Lists configuration doradus.yaml settings, including value overridden by runtime arguments.
`StartTime`	Provides the start time of the Doradus server as a `Date.getTime()` value.
`Throughput`	Provides a histogram of REST API request response times in seconds for various sampling rates (e.g., 1 minute, 5 minutes, 15 minutes.)
`WorkingDirectory`	The working directory of the Doradus server.

5.3.2. StorageManager

`StorageManager` is operative only when the Task Manager service has been initialized (`com.dell.doradus.service.taskmanager.TaskManagerService`). This is always the case when the server is started as a standalone process but optional when Doradus is embedded within another application. `StorageManager` provides the following attributes:

Attribute Name	Description
AppNames	The Doradus application names known to the Task Manager.
GlobalDefaultSettings	The global task schedule settings, which are used for tasks that do not have a more specific (e.g., application- or table-specific) schedule default. The global schedule is usually "never", meaning no default schedule is applied to tasks.
JobList	The list of tasks currently executing.
NodesCount	The number of Cassandra nodes in the cluster.
OS	The operating system of the Doradus server.
OperationMode	The operation mode of the first connected Cassandra mode (NORMAL, LEAVING, JOINING, etc.)
RecentJob	Attributes about the most recent job executed through the StorageManager interface.
ReleaseVersion	The Doradus server release version.
StartMode	The execution mode of the first connected Cassandra node: 1 (FOREGROUND), 2 (BACKGROUND), or 0 (UNKNOWN).

The StorageManager MXBean also provides operations that perform basic administrative functions, summarized below:

Operation Name	Description
getJobByID	Get the status of a job with a specific ID.
startCreateSnapshot	Start a task to create a backup snapshot of the first Cassandra node connected to the Doradus server, assigning the snapshot a name.
startDeleteSnapshot	Delete a backup snapshot with a given name.
startRestoreFromSnapshot	Restore the Cassandra node connected to the Doradus server from a given snapshot game.
sendInterruptTaskCommand	Interrupt an executing task with a given name, belonging to a given application.
sendSuspectSchedulingCommand	Suspend the scheduling of a task with a given name, belonging to a given application.
sendResumeSchedulingCommand	Resume the scheduling of a task with a given name, belonging to a given application.
getAppSettings	Get the default task schedule settings for a given application name.
getTaskStatus	Get the status of a task with a given name, belonging to a given application.

6. Cassandra Configuration and Operation

This section describes the key configuration optional available for Cassandra. Not all options are covered – only common options and those that warrant modification in certain situations. Note that Cassandra options change from release to release. The options described here are relevant to the Cassandra 2.x release. Also, only options relevant to Windows installations are described.

6.1. Cassandra Script Files

Within the folder `{cassandra_home}/bin`, the following are the key scripts that can be used to launch Cassandra and various tools:

- **`cassandra.bat`**: This is the script file bundled with Cassandra to launch it from a command console.
- **`cassandra-cli.bat`**: This script file runs the Cassandra command line interface (CLI) tool. It can be used to perform low-level queries and to modify certain Cassandra schema options.
- **`nodetool.bat`**: This script launches the Cassandra nodetool utility, which can perform various Cassandra monitoring and operational commands: determining cluster status, forcing compaction, rebuilding a node, etc.

The bin folder contains .bat files for other tools as well. Except for `cassandra.bat`, most of the script files can run with the parameter "-?" to get help information.

6.1.1. Setting Cassandra Runtime Memory

By default, Cassandra is set to use only 1GB of memory. This is OK for testing on a developer workstation, but for production work, 8GB to 16GB of memory is recommended. This is controlled by the `-Xms` and `-Xmx` parameters. Within the `{cassandra_home}/bin/cassandra.bat`, these values are explicitly defined and can be modified as shown below:

```
set JAVA_OPTS=^
 -Xms8G^
 -Xmx8G^
 ...
```

For use by Doradus, 8GB is the recommended minimum memory. If the machine has 32GB of physical memory or more, 16GB is the recommended memory allocated to Cassandra.

6.1.2. Disabling Assertions

The Java runtime option `-ea` (or `-enableassertions`) is used to enable diagnostic "assert" statements in Java applications. It is useful in development/debugging environments since additional internal checking is performed, but it has a small performance cost. It should be disabled in production environments to eliminate the performance cost. This can be done by simply removing the `-ea` option in JVM options or replacing it with `-da` (or `-disableassertions`). In both the Cassandra script files `{doradus_home}/bin/doradus-cassandra.bat` and `{cassandra_home}/bin/cassandra.bat`, assertions are declared in the `set JAVA_OPTS` statement as shown below:

```
set JAVA_OPTS=-ea^
...
```

6.2. Cassandra Configuration Files

The following are the primary Cassandra configuration files in the `{cassandra_home}/conf` folder:

- **`cassandra.yaml`**: This file provides global configuration options. There are a lot of options in this file, but only a few typically require modification from their defaults.

- **`log4j-server.properties`**: This file controls logging features of the Cassandra instance.

The following sections describe the options most commonly modified.

6.2.1. Setting Cassandra Logging Options

Cassandra logs messages about its operation using the log4j logging facility. The file that controls logging options is `{cassandra_home}/conf/log4j-server.properties`. The one option you should change is shown below:

```
log4j.appender.R.File=/var/log/cassandra/system.log
```

You should change this line to use a valid folder path where logs will be kept. The initial log file will be called `system.log`. When a log file becomes full (default is 20MB), it is renamed with a numeric suffix (e.g., ".1", ".2"). When 50 files are created, the oldest files are removed.

The other option you may want to modify under specific situations is:

```
log4j.rootLogger=INFO,stdout,R
```

This line sets the global logging level to INFO. In certain diagnostic situations, you might want to change the INFO keyword to DEBUG to generate additional logging messages. Note that DEBUG log level causes log files to grow quickly and slow down Cassandra.

6.2.2. Setting Cassandra Data File Locations

Cassandra creates three kinds of data files: commit logs, SSTables, and saved caches. (Somewhat confusingly, the SSTable files are sometimes called "data files" even though all three kinds of files hold data.) The folder path of each file kind is defined in `cassandra.yaml`. You should change all of the following folder options:

```
commitlog_directory: /var/lib/cassandra/commitlog
```

Set this option to the folder where commit logs should be stored. It should be a different disk than any of the SSTable disks (see below). Multiple commit logs are created in this folder, but they are deleted when they become obsolete, so typically commit logs do not require a lot of space.

```
data_file_directories:
    - /var/lib/cassandra/data
```

Set this option to at least one root folder where SSTable files are to be stored. SSTables are the primary files containing application data. Each folder is listed on a secondary line, indented and beginning with a dash. Multiple data folders are recommended for better performance (see below).

```
saved_caches_directory: /var/lib/cassandra/saved_caches
```

Set this option to a valid folder name where Cassandra will save key and row caches that it builds. It can be the same disk as the commit log or where software is installed, but it shouldn't be one of the SSTable disks. The size of disk space for caches depend on cache option settings.

When updates are sent to Cassandra, they are first written to a commit log file. The commit files are "replayed" when a restart occurs, thereby providing recovery for updates that may not have been written to an SSTable file. Because commit logs are removed when they are no longer needed, they typically do not use much disk space.

After updates are written to the commit log, they are stored in memory and eventually sorted and flushed to disk as SSTables. Each SSTable is represented by multiple files including data, hash, and index files. When Cassandra is configured with multiple data file directories, it flushes each SSTable to the directory that has the most available space. Therefore, best practices for the commit log and SSTable files are:

1. Each SSTable folder should reside on a separate disk. This allows concurrent I/Os: a separate I/O can be initiated for each disk.

2. Each SSTable disk should be of the same size and used solely for SSTables. This prevents disk contention with other files, and it allows all disks to grow at the same rate.

3. The commit log folder should reside on its own disk. Because data is flushed quickly as it is received, the commit log folder can receive a high volume of I/O, hence it should use its own disk to prevent contention with SSTable files. The disk does not have to be large since commit logs are discarded fairly quickly.

6.2.3. Cassandra Security Options

As described in the Security Considerations section, Cassandra does not encrypt its commit log or data files. Disk files must be protected through operating system-level file security or encryption (e.g., Windows EFS). This section describes how to protect Cassandra's communication protocols: Thrift, CQL, JMX, and Gossip.

6.2.3.1. Securing the Cassandra API

Either the Cassandra Thrift or CQL API can be used by Doradus. By default, both APIs use an unencrypted connection and allow any process to connect and authenticate. To prevent unauthorized applications from directly accessing Cassandra, you can enable TLS.

The general steps for enabling TLS are described below:

1. In the `cassandra.yaml` file on each Cassandra node: enable TLS by setting `enabled` to `true` under `client_encryption_options`. Require client authentication by setting `require_client_auth` to `true`. When client authentication is enabled, the `truststore` and `truststore_password` options must also be set. Finally, `cipher_suites` should be set to one or more cipher suites that are accessible to the JRE. An example of the required settings is shown below:

```
client_encryption_options:
    enabled: true
    keystore: ../conf/.keystore
    keystore_password: cassandra
    require_client_auth: true
```

```
truststore: conf/.truststore
truststore_password: cassandra
cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA]
```

2. Create a certificate that will be used by Doradus and add it to the Cassandra truststore on each node.

3. In the doradus.yaml file for each Doradus instance: enable TLS by setting `dbtls` to `true`. Finally, set `dbtls_cipher_suites` to the same cipher(s) defined for Cassandra. An example of these settings is shown below:

   ```
   dbtls: true
   dbtls_cipher_suites: [TLS_RSA_WITH_AES_128_CBC_SHA]
   ```

4. Add the certificate created in step 2 to the keystore for each Doradus instance. This requires that the `keystore` and `keystorepassword` in each doradus.yaml file is also set.

More information about enabling TLS and creating certificates can be found in documentation such as the following:

http://www.datastax.com/documentation/cassandra/2.0/cassandra/security/secureSSLClientToNode_t.html

6.2.3.2. Securing the Cassandra JMX Protocol

Cassandra supports the Java Management Extensions (JMX) protocol for monitoring and controlling Cassandra processes. JMX can be secured with authorization and/or encryption, however, JMX security is disabled in the `{cassandra_home}/bin/cassandra.bat` file that is included with Doradus. The common JMX options are defined in the JAVA_OPTS environment variable as shown below:

```
REM ***** JAVA options *****
set JAVA_OPTS=-ea^
 ...
 -Dcom.sun.management.jmxremote.port=7199^
 -Dcom.sun.management.jmxremote.ssl=false^
 -Dcom.sun.management.jmxremote.authenticate=false^
 ...
```

These options are summarized below:

- **`port`**: This sets the port number that the remote JMX clients must use. As shown, port 7199 is the default used for Cassandra.

- **`ssl`**: When set to true, this option requires remote clients to use SSL (TLS) to connect to the JMX port. When SSL is enabled, additional options are available to require remote clients to have a client-side certificate.

- **`authenticate`**: When set to true, this option requires remote clients to provide a user ID and password in order to connection. Additional parameters are required to define allowed user IDs and passwords and the locations of the corresponding files.

Because JMX is a standard and external documentation is available for securing JMX, details for using SSL and/or authentication are not covered here. See for example the following:

http://docs.oracle.com/javase/6/docs/technotes/guides/management/agent.html

Though not shown above, here is another useful option: On multi-homed systems, the define `java.rmi.server.hostname` can be set to cause JMX to bind to a specific IP address instead of the default one. For example:

```
set JAVA_OPTS=-Djava.rmi.server.hostname=10.1.82.121^
...
```

This causes the JMX port to bind to address 10.1.82.121.

6.2.3.3. Securing the Cassandra Gossip Protocol

In a multi-node cluster, each Cassandra node communicates with peer nodes using the Gossip protocol. For non-encrypted connections, the Gossip protocol uses a TCP port defined by the following `cassandra.yaml` option:

```
storage_port: 7000
```

When SSL is enabled for the Gossip protocol, the following `cassandra.yaml` file option defines the port number used:

```
ssl_storage_port: 7001
```

All nodes in a cluster should be configured to use the same `storage_port` and `ssl_storage_port`. To prevent eavesdropping or unauthorized disruptions, the gossip protocol should be secured in production environments. However, because the protocol is used for high-performance operations such as replicating data between nodes, encryption is not recommended except for communication between remote locations.

For co-located nodes, the easiest way to secure the Gossip API is to deploy all Cassandra nodes on the same subnet and disallow access to the Gossip port from outside the subnet.

In large Cassandra deployments where multiple "racks" or "data centers" are deployed, each having some number of Cassandra nodes, the Gossip protocol can be secured for cross-rack or cross-data center communication. This is done with the following options in the `cassandra.yaml` file:

```
encryption_options:
    internode_encryption: none
    keystore: conf/.keystore
    keystore_password: cassandra
    truststore: conf/.truststore
    truststore_password: cassandra
```

Internode encryption (over the Gossip API) is enabled or disabled by the setting of the `internode_encryption` option. The following options are recognized:

- `none`: This disables all inter-node encryption, meaning Cassandra nodes use unencrypted communication using the defined `storage_port`.
- `all`: This enables encryption for all inter-node communication using the defined `ssl_storage_port`.
- `rack`: This uses non-encrypted communication for nodes defined to be in the same rack (cabinet) and encrypted communication between nodes defined to be in different racks.

- `dc`: This uses non-encrypted communication for nodes defined to be in the same data center and encrypted communication between nodes defined to be in different data centers.

When any encryption is enabled for the Gossip protocol, all authentication, key exchange, and data transfer occurs with TLS v1 using RSA 1024 bit keys. This encryption suite is referred to as TLS_RSA_WITH_AES_128_CBC_SHA. This requires that `keystore` and `truststore` files are defined and initialized. These files are password-protected using the `keystore_password` and `truststore_password` options. Instructions for creating these files can be found publicly, such as in this link:

http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html#CreateKeystore

6.2.4. Configuring Cassandra for Clusters

By default, Cassandra assumes that it is operating as a stand-alone node. It must be configured to operate in a cluster. The following `cassandra.yaml` options affect a node's participation in a cluster:

- `cluster_name`: All nodes in the cluster must have the same name, which differentiates the cluster from other nodes that might be working in the same network or even on the same machine. The default name is "Test Cluster", so you should change this to something else like "Doradus Cluster".

- `initial_token`: This value defines the beginning range of key values for which the node will be the primary owner. It is not set by default, and it *may* be valid to leave it unset when configuring a new node. However, for a "balanced" cluster, you will need to set this value for each node.

- `seeds`: Seeds are IP addresses of neighboring nodes that this node can contact using the gossip protocol. The addresses provide only an initial set: after a node is running, it will memorize the addresses of other nodes in the network and contact them when necessary. The seeds are therefore necessary for the initial execution of a new node. Cassandra provides a generalized "seed provider" interface, but the built-in "simple seed provider" is sufficient for most situations.

- `listen_address`: This is the IP address that tells other nodes what IP address to use to communicate to this node. To participate in a cluster, you must change this from its default of "localhost". A host name can be used but is not recommended. The "any address" 0.0.0.0 will not work. You should use a static IP address visible to all other nodes. If the machine is multi-homed, a non-externally visible address (192.x or 10.x) is a good choice.

- `partitioner`: Beginning with the Cassandra 1.2 release, the default for this parameter is now `Murmur3Partitioner`. This random partitioning algorithm is more efficient than the older `RandomPartitioner` scheme, although the two are incompatible. All nodes in the cluster should use the same partitioning scheme. If you upgrade from an older Cassandra release, you'll need to ensure this parameter matches your existing value.

For more details on Cassandra configuration options, see http://wiki.apache.org/cassandra/Operations. The Wiki site also has information on topics such as:

- Adding new nodes to an existing cluster

- Migrating from the older to newer random partitioning scheme

- Recovering a node that has died

- Removing a node from the cluster
- Changing a cluster's replication factor
- Deploying larger clusters within multiple *racks* (cabinets) and even *data centers*

In addition to the Wiki site, there are several online sources and books on Cassandra configuration such as:

- The Cassandra web site http://cassandra.apache.org/ and especially the Operations page: http://wiki.apache.org/cassandra/Operations
- The Planet Cassandra user community site: http://planetcassandra.org/
- The Datastax web site http://www.datastax.com/ and its online documents.
- Books such as *Cassandra: The Definitive Guide* and *Cassandra High Performance Cookbook*

6.2.5. Other Cassandra Configuration Options

In addition to the options described in this section, there are other options in the `cassandra.yaml` file that you might want to change in certain circumstances. Here a list of the most common options:

- `concurrent_reads`: This value controls how many outstanding read operations are allowed at once. A recommended value is 16 times the number of data disks used.
- `concurrent_writes`: This value controls how many outstanding write operations are allowed at once. A recommended value is 8 times the number of cores present on the machine.

The next option is common to both the Thrift and CQL APIs:

- `rpc_address`: This value controls the address(es) to which Cassandra binds when listening for client connections. The same value is used to control both the Thrift and CQL APIs. The value `localhost` will allow only local connections. A specific IP address can be used, or the address `0.0.0.0` can be used to cause Cassandra to accept connections on all network interfaces.

The next options are specific to the Thrift API:

- `start_rpc`: When `true`, this option causes Cassandra to initialize the Thrift API.
- `rpc_port`: This is the Thrift API port that applications such as Doradus connect to. You can change it from its default of `9160`, but it should be the same on all nodes. (And you must configure Doradus to know what port to use.)
- `thrift_framed_transport_size_in_mb`: This value controls the maximum size of a Thrift message that Cassandra will accept. The default (16MB) is often too small for Doradus applications that use "batch updates". A good idea is to increase this to 160MB.

The next options are specific to the CQL API:

- `start_native_transport`: When this option is `true`, Cassandra initializes the CQL API.

- `native_transport_port`: This is the CQL API port that applications such as Doradus connect to. You can change it from its default of `9042`, but it should be the same on all nodes. If you configure Doradus to use the CQL API, its configuration must match this value.

6.3. Monitoring Cassandra

Cassandra supports the JMX protocol to provide monitoring capabilities. In addition to the normal Java parameters that can be monitored via JMX (memory usage, CPU usage, classes loaded, etc.), Cassandra implements numerous additional MBeans ("management beans"), which provide insights into core Cassandra operations and status. A detailed description of the Cassandra-specific MBeans can be found here:

http://wiki.apache.org/cassandra/JmxInterface

Generic JMX tools can be used to monitor Cassandra, including the JConsole tool that is bundled with the Java JDK. To use JConsole, install the Java SE 6 JDK and run `jconsole.exe` from the bin directory. JConsole will automatically find all Java processes on the current machine and offer to connect to them. If the Cassandra process you want to monitor is executing on a remote machine, select the Remote Process button and enter the host name or IP address and JMX port number of the remote Cassandra process you want to connect to. Example:

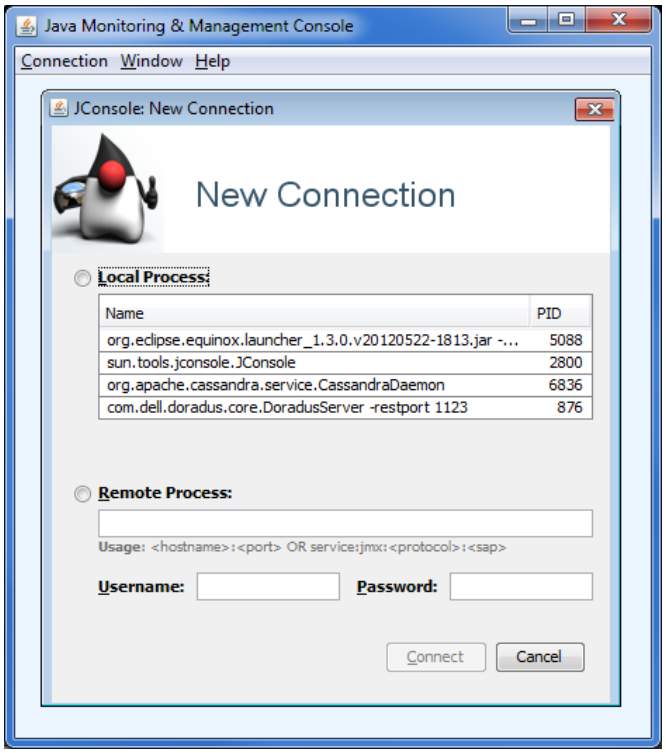

Figure 9 - JConsole New Connection Dialog

If the remote JMX process is secured with authentication, enter the Username and Password with which JMX has been configured. When JConsole connects, it offers several panes for examining generic Java metrics as well as the Cassandra-specific MBeans. Below is an example of the Overview pane for a Cassandra process:

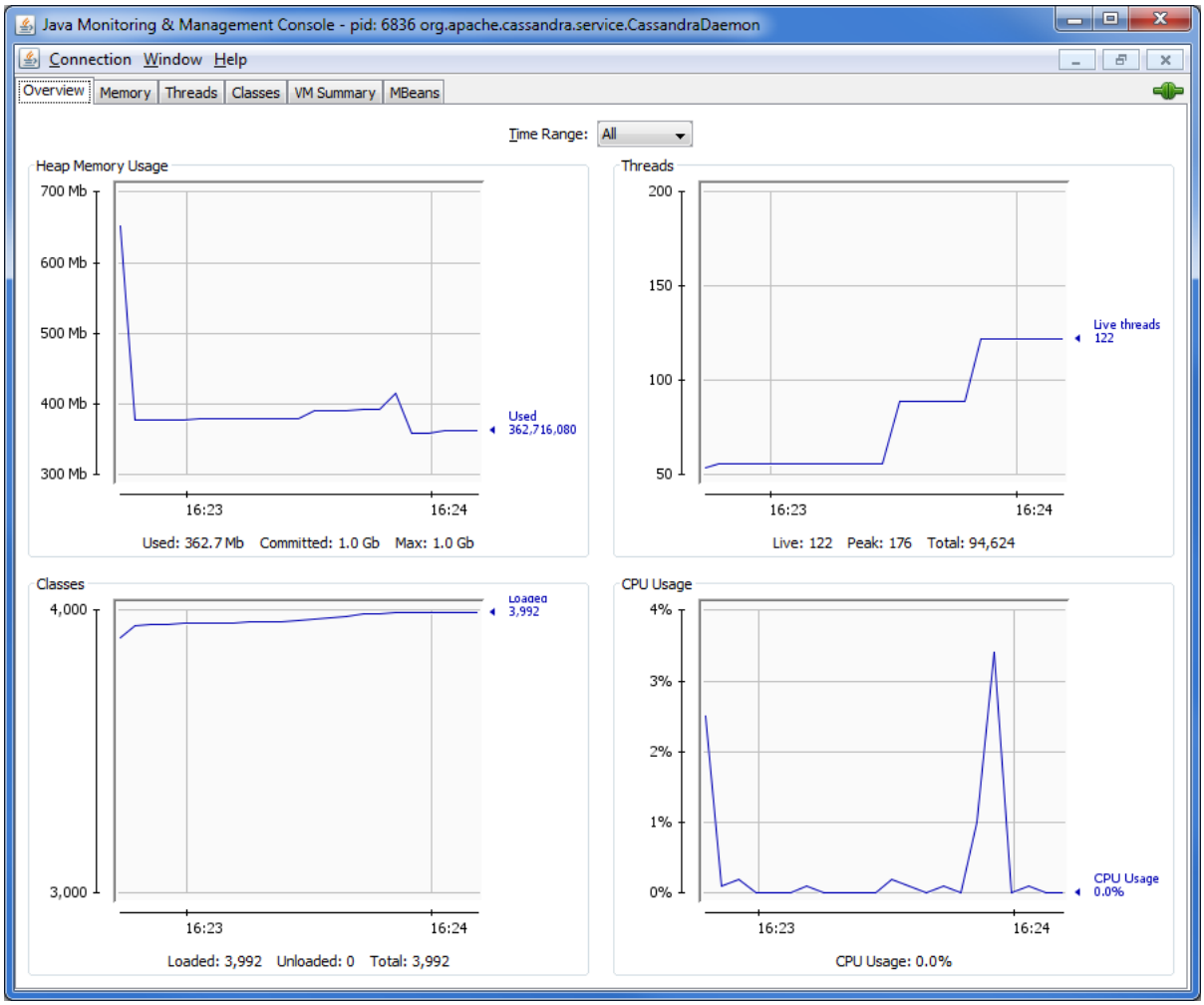

Figure 10 - JConsole Overview Pane

Although JConsole can access Cassandra-specific MBeans, it provides generic UI widgets for accessing these functions. Furthermore, although JConsole can connect to multiple JMX-enabled processes at once, it does not provide integrated, cross-process metrics.

To get more detailed information about Cassandra information from a cluster perspective, there are a number of third party tools becoming available. See, for example, OpsCenter, which is available from Datastax:

http://www.datastax.com/