vmware vFabric GemFire

ELASTIC DATA MANAGEMENT

FOR

VIRTUALIZED AND CLOUD-BASED APPLICATIONS

Data Integrity

VMware, Inc.

3401 Hillview Ave
Palo Alto, CA 94304 USA
Tel: 1-877-486-9273
Fax: 650-427-5001

Introduction

Loss of data can affect your business in many ways causing substantial, and possibly irrevocable, harm. The causes of data loss can take many forms: hardware failure, hardware destruction, software failure and the list can go on. If any one of these incidents occur, the value of lost productivity and diagnostics would be overshadowed by the risk to sales, irretrievable data, regulation compliancy, project failure and a possible lawsuit.

Many distributed data solutions exist today and often their capabilities are incorporated into middle tier platforms. However, these implementations tend to suffer from being too relaxed with respect to data consistency, lack of support for managing data integrity, or are limited by the available process memory of a single machine. The design premise of GemFire is to distribute data across a cluster of nodes, providing a choice of full replication to selected nodes or fault tolerant partitions of data across selected nodes while maintaining your data's integrity.

Without data integrity, you cannot have assurance that your data is consistent and can be counted on to make important business decisions. In this example, we will show you how GemFire can help mitigate some of the risks associated with data loss and maintain the integrity of your data. We will model two failure cases to highlight some of the High Availability and Data Integrity capabilities of GemFire. At the conclusion, GemFire will have survived or recovered from these types of possibly fatal system occurrences.

In the first scenario, we will pull the plug on one of the GemFire data nodes and continue updating some of our data. Our downed member will have out of date information, which would be catastrophic to our data integrity if this non-operational data were to be reintroduced. We will restart the entire GemFire data fabric as a way to stress this capability. If we only restart the downed member the remaining members would populate it on the rejoin, the same as adding new capacity to your data fabric.

For the second type of failure, we will corrupt the data files in our system while we pull the plug on our GemFire data node. When we restart, GemFire will alert the system administrator and the troubleshooting will begin. However, since GemFire is already keeping redundant copies of your data secure on the remaining nodes, the system administrator only has to do basic certification of the OS and hardware. Once that is completed, only a simple GemFire command is needed to allow the other data nodes to know that this system is safe and operational. Of course if it is more cost effective, a standby system can hot swapped in as in the first scenario and the failed box sent back to the labs.

Installation

To install the example, just unzip it to a convenient location. Take care to make sure the folder names in the install path do not have any spaces in them. Also note that GemFire will be writing to that directory for its persistence, so ensure the directory is also writeable. In the examples that follow I have unzipped the examples to C:\dev\Data_Integrity.

This example also requires Java 1.6 or better to be installed. Java can be downloaded from http://java.sun.com/j2se.

If you already have java installed you can verify its version by running the `java -version` command.

```
java -version
java version "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03)
Java HotSpot(TM) Client VM (build 20.1-b02, mixed mode, sharing)
```

Figure 1 Checking the installed version of java,

Once Java has been installed the example scripts expect the JAVA_HOME environment variable to point to the root location of the JDK installation directory. If you are not able to change the path environment variable you can edit the `bin/setGemFireEnv` (linux) or `bin/setGemFireEnv.bat` (windows) to set the location of java on your machine.

Contents of the bin directory (with the .bat or .sh stripped off for brevity):

- `cacheLoader` – Launches a java program that loads data for verification into GemFire. See `src/demo/vmware/gemfire/poc/CacheLoader.java` for more details.
- `clearGemFire` – Deletes all of the GemFire data pertaining to this example. Only run this command when all of the GemFire processes are shutdown (see stopGemFire).
- `corruptData` - This script takes an argument which is the server number to corrupt. For this example valid range is 1-3.
- `revokeDiskStores` – Calls GemFire to see if there are any missing disk stores. If there is any missing it will revoke them so the cluster will start up.
- `setGemFireEnv` – Sets up the environment for the currently running script. This script is meant to be sources from within another script, do not run it alone. If run alone it will error out.
- `startCacheServer` – Allows the user to easily start a given server. For this example valid arguments are 1-3.
- `startGemFire` – Starts the GemFire locator and 3 GemFire data management nodes.
- `stopCacheServer` – Allows the user to easily stop a given server. For this example valid arguments are 1-3.

- `stopGemFire` – Connects to the GemFire fabric and requests that all the machines be shutdown. Then it stops the locator process.
- `validator` – Once the `cacheLoader` program has finished this program will verify the contents of the data fabric match expected values. See `src/demo/vmware/gemfire/poc/Validator.java` for more details.

Each one of these scripts is considered to be an ease of use script. It is highly recommended to review these scripts so you can model a GemFire deployment using your own automated tool set.

All of the source code of this example can be found in the `src` directory.

GemFire Concepts

Since GemFire Enterprise Data Fabric has some unique terms it might be helpful if I discuss some of them before we get started.

Region –A logical grouping within the data fabric for a single data set. You can define any number of regions within your fabric. Each region has its own configurable settings governing such things as the data storage model, local data storage and management, partitioning, data event distribution, and data persistence.

A Region is the core of GemFire, and is the main API that is used when developing with GemFire. A Region implements the `java.util.concurrent.ConcurrentMap,` which is also an extension of `java.util.Map`. This means that if your developers can use a hash map, your developers can use GemFire.

Partitioning – A logical division of a region that is distributed over members of the data fabric. For high availability, configure redundant copies so that each data bucket is stored in more than one member, with one member holding the primary copy.

Bucket - A unit of storage for distribution and replication, which is distributed in accordance to the region's attribute settings.

Server – This is what provides all of the form and function that makes up GemFire.

Client / Client Cache - The client enables any Java, C++, or C# application to access, interact with, and register interest in data managed by a Data Management Node. Clients can also be embedded within a Java application server's process to provide HTTP session replication or Hibernate L2 object caching with eviction expiry and overflow to disk.

Serialization – The process in which an object state can be saved, transmitted, received and restored. The process of saving or transmitting state is called serializing. The process of receiving and restoring state is called deserialization.

Locator – A GemFire locator is a registry where servers can advertise their location so clients and other servers can discover the active servers.

Cache.xml – This allows system architects to declaratively create the cache through xml. While the file can be any name it commonly refered to as a "cache.xml" file.

Cache – Is a set of Regions. This is also referred to as a data fabric.

Single Hop Optimization - the client maintains a set of meta-data of where a partitioned region's data is hosted in the data fabric. To access a single entry, the client directly contacts the server that hosts the key—in a single hop.

If we turn off single hop optimization, the client uses whatever server connection is available. The server that receives the request determines the data location and contacts the host, which might be a different server. With single hop disabled, more multiple-hop requests are made to the server system

Redundancy - Highly available partitioned regions provide reliability by maintaining redundant data. When you configure a partitioned region for redundancy, each entry in the region is stored in at least two members' caches. If one of its members fails, operations continue on the partitioned region with no interruption of service. Recovering redundancy can be configured to take place immediately, or delayed for a configurable amount of time until a replacement system is started.

Without redundancy, failure of any of the members hosting the partitioned region results in partial loss of data. Typically redundancy is not used when applications can directly read from another data source, or when write performance out weighs read performance. The addition of a redundant copy ensures high availability of your data in the partitioned region.

Failover - When a server hosting a subscription queue fails, the queueing responsibilities pass to another server. How this happens depends on whether the new server is a secondary server. In any case, all failover activities are carried out automatically by the GemFire system.

Online Material

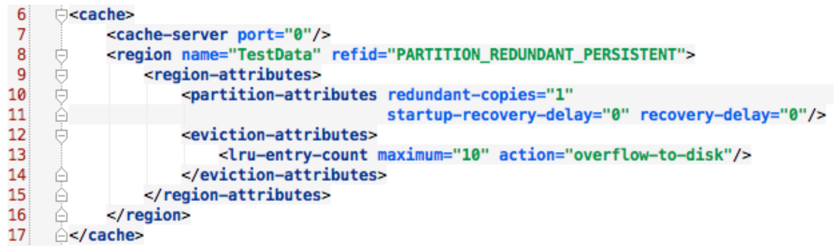
GemFire also has a strong online community where you can find:

- Documentation - https://www.vmware.com/support/pubs/vfabric-gemfire.html
- Forums - http://communities.vmware.com/community/vmtn/appplatform/vfabric_gemfire

Configuration

There are a large number of configuration options to enable GemFire to match your data strategy. Our example only touches on a few of them. If you are interested in reading more on how GemFire can be configured to suit your needs please visit our online documentation, or contact your vFabric sales representative.

Server Configuration



```
<cache>
    <cache-server port="0"/>
    <region name="TestData" refid="PARTITION_REDUNDANT_PERSISTENT">
        <region-attributes>
            <partition-attributes redundant-copies="1"
                                  startup-recovery-delay="0" recovery-delay="0"/>
            <eviction-attributes>
                <lru-entry-count maximum="10" action="overflow-to-disk"/>
            </eviction-attributes>
        </region-attributes>
    </region>
</cache>
```

Figure 2 Declarative server.xml file.

This is a typical cache.xml file. For our server we have specified region that is partitioned across the fabric called `TestData`. Lets go over all of the attributes that we have placed on the region.

Partition – Our data is going to be spread over the data fabric. The location of the primary bucket is by default found using the hash of the key modulus the number of buckets the fabric supports. Modulus is the remainder after dividing by a number.

Example: By default there are 113 buckets in a partitioned region. Let say a key's hash code is 256. Then the value will be placed in bucket 30 (256 mod 113 = 30).

Redundant - There will be 0-3 copies of a given bucket in this region. In this example we have set redundancy to 1. Which means there will be at least 2 copies of every bucket spread across the data fabric.

Recovery Delay – This is the time GemFire will wait before it attempts to recover from a failure case. Our settings of zero mean that GemFire will rebalance the partition as soon as a member is discovered to be down. Also Gemfire will rebalance as soon as a new member joins a system.

Having GemFire instantly recover might not be the best option for every circumstance. A perfect example is rolling out a patch. During a OS or Application patch cycle GemFire would be brought down. This would be the same as a node failure and would cause a large amount of network and system utilization as it attempts to recover. One time to bring down the system so it can be patched, another time while its being brought back into service.

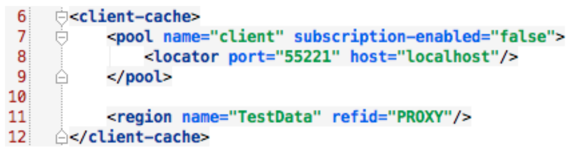
Eviction – Here we can specify what to do when a region starts growing. GemFire currently supports three different eviction algorithms:

1. Number of Objects in a Region
2. The size of the Region
3. The size of the Java VM heap that the Region is running in.

For our example we have chosen a really basic scheme where only 10 objects are kept in memory. The remaining objects are over flown to disk.

Client Configuration

Clients typically connect to a locator to discover the connection information for servers to attach to. For each request the locator responds with the connection information for the least loaded server. At that time any client API calls that aren't satisfied by the clients local cache are forwarded to the servers.



```
<client-cache>
    <pool name="client" subscription-enabled="false">
        <locator port="55221" host="localhost"/>
    </pool>

    <region name="TestData" refid="PROXY"/>
</client-cache>
```

Figure 3 Declarative client.xml file.

Our client cache is really basic, lets examine the configuration in detail.

Pool – We have a server connection pool defined by the name of `client`. The pool name can later be used in the client code to find the Pool of connections to a set of servers.

Subscription Enabled - You can configure your servers to automatically push cache updates to clients. To do this, create your client Pool with subscription-enabled set to true and register client interest in events on the server. Since we are only interested in basic put and get operations we have set the subscription to false.

Region – Here we specify that we are using the `TestData` region. Since we have configured this region as a PROXY region our client does not hold any data locally and must rely on the servers for all data.

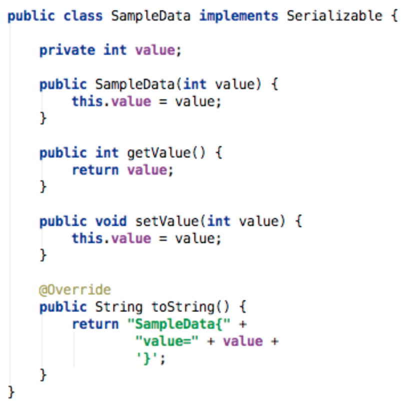
Code

This example has three classes:

1. `SampleData` - A basic object that will be stored into a partitioned region within GemFire
2. `CacheLoader` - Loads the data fabric with a known data pattern that can be easily verified.
3. `Validator` - Verifies that the content of the Data Fabric is what it should be.

Lets step through the code so we can see how GemFire is used.

SampleData



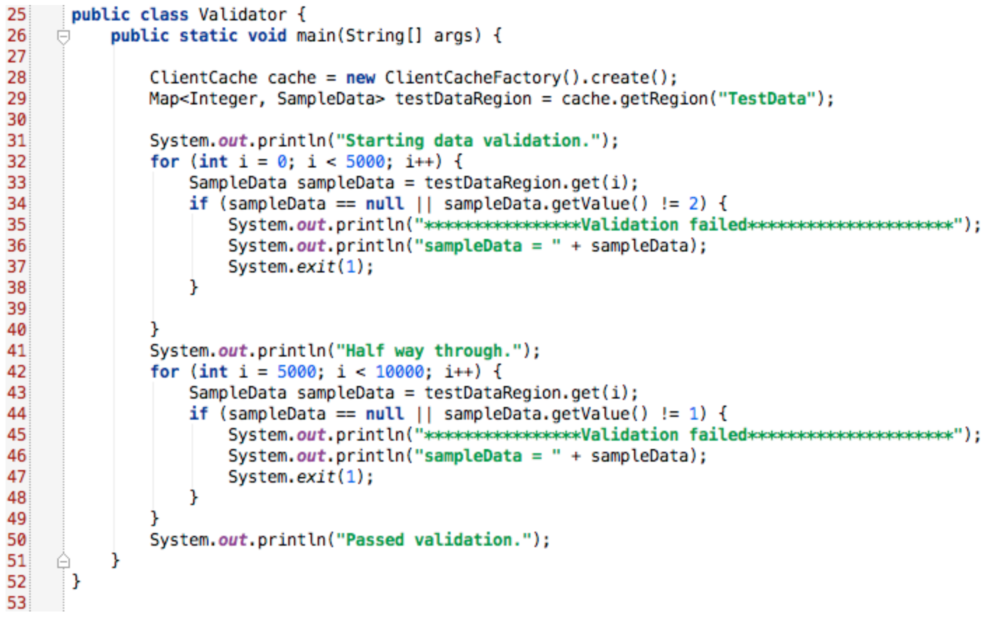
```java
public class SampleData implements Serializable {

    private int value;

    public SampleData(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "SampleData{" +
                "value=" + value +
                '}';
    }
}
```

Figure 4 SampleData Class.

Since this class will have its instance data stored in GemFire, we have marked the class as implementing `Serializable`. `Serializable` is a standard Java interface that Java knows how automatically to serialize and de-serialize objects that implemented this interface.

In an effort to keep this example as simple as possible, `SampleData` has only a single variable. This variable will allow us to determine if the data fabric has consistent state when we are done.

Validator



```
public class Validator {
    public static void main(String[] args) {

        ClientCache cache = new ClientCacheFactory().create();
        Map<Integer, SampleData> testDataRegion = cache.getRegion("TestData");

        System.out.println("Starting data validation.");
        for (int i = 0; i < 5000; i++) {
            SampleData sampleData = testDataRegion.get(i);
            if (sampleData == null || sampleData.getValue() != 2) {
                System.out.println("****************Validation failed*********************");
                System.out.println("sampleData = " + sampleData);
                System.exit(1);
            }

        }
        System.out.println("Half way through.");
        for (int i = 5000; i < 10000; i++) {
            SampleData sampleData = testDataRegion.get(i);
            if (sampleData == null || sampleData.getValue() != 1) {
                System.out.println("****************Validation failed*********************");
                System.out.println("sampleData = " + sampleData);
                System.exit(1);
            }
        }
        System.out.println("Passed validation.");
    }
}
```

Figure 5 Validator Class.

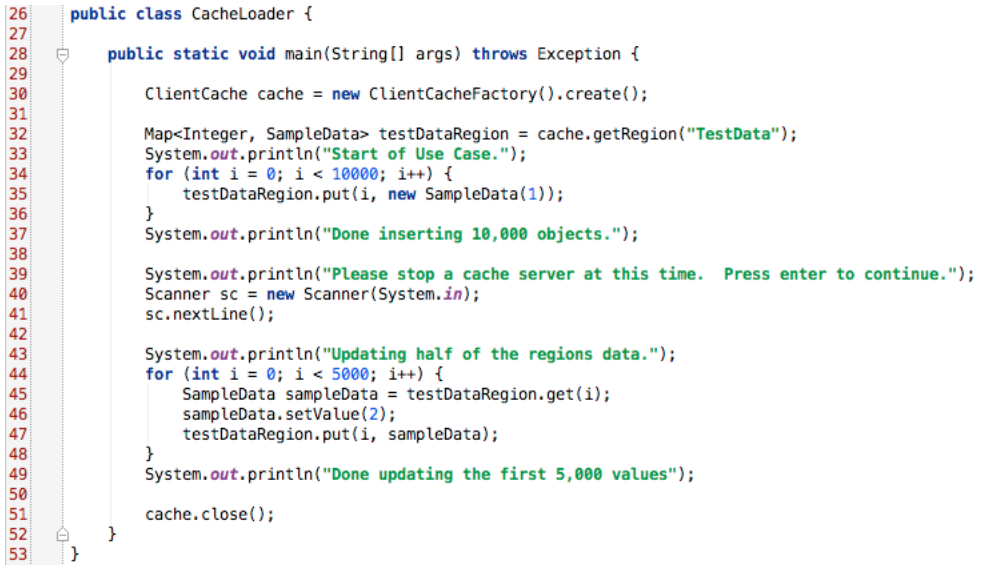
While we are not using some of the more advanced features in this example, it does show the basic operating principals of a GemFire application. From this code you can see how easy it is to integrate GemFire into your code baseline. I even up casted the `Region` to a standard Java `Map` to put emphasis on the simplicity of GemFire.

The basic difference between `Region` and the standard Java `Map` is GemFire subscription and event handling. Subscription handling means we can register interest in a subset, or all, of what is hosted in the `Region`. Event handling is where we can add `Region` listeners. `Region` listeners allow subscribers to know when important data events happen such as add, update, delete or invalidate.

Walkthrough of the Validator class:

> We attached our client to the GemFire data fabric and grab our data `testDataRegion` from there. Then we start the validation. First we iterate over the first 5,000 SampleData objects assuring that their value is 2. If we have a deviation then we inform the user and exit with an error code. Once we are half way through we iterate over the next 5,000 SampleData objects assuring that their value is 1. If there is deviation then we also exit with an error code. Also note that if we ever fail to retrieve a value, it is also an error condition.

CacheLoader



```java
public class CacheLoader {

    public static void main(String[] args) throws Exception {

        ClientCache cache = new ClientCacheFactory().create();

        Map<Integer, SampleData> testDataRegion = cache.getRegion("TestData");
        System.out.println("Start of Use Case.");
        for (int i = 0; i < 10000; i++) {
            testDataRegion.put(i, new SampleData(1));
        }
        System.out.println("Done inserting 10,000 objects.");

        System.out.println("Please stop a cache server at this time.  Press enter to continue.");
        Scanner sc = new Scanner(System.in);
        sc.nextLine();

        System.out.println("Updating half of the regions data.");
        for (int i = 0; i < 5000; i++) {
            SampleData sampleData = testDataRegion.get(i);
            sampleData.setValue(2);
            testDataRegion.put(i, sampleData);
        }
        System.out.println("Done updating the first 5,000 values");

        cache.close();
    }
}
```

Figure 6 CacheLoader class.

Just like in the `Validator` code above, I up casted the `Region` to the `Map` interface. Again this is to highlight what your developers will have to work with when integrating GemFire into to your solution.

Walkthrough of the CacheLoader class:

First we create the client cache that reads our configuration files and attaches it to the GemFire data fabric. We then look up our `Region` and put 10,000 `SampleData` objects into that region. Each `SampleData` is initialized with the value one and its key is the order in which is was created . We then pause for a user to press the `<enter>` key.

Loop over the first 5,000 key and get the corresponding values from the servers. We then change the value of our `SampleData` so it contains the value ‘2’. Then we put the updated object back into the distributed system.

Execution

In this example we will show screen shots from Windows 7, however the example can be run on Linux as well. There is a difference in how the two operating systems call their command-line interface. In Windows it is called a command prompt, where as in Linux it is called a terminal. For this document I will refer to them as a command prompt.

For this example I will be using Windows 7 if you are using a different OS version or distribution your mileage may vary.

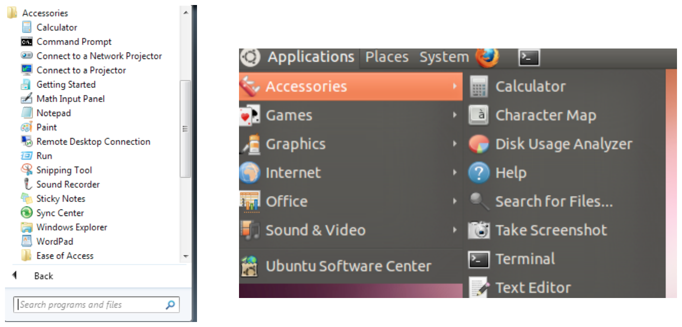


Figure 7 Location of Command Prompt on Windows and Linux

Rejoining Scenario

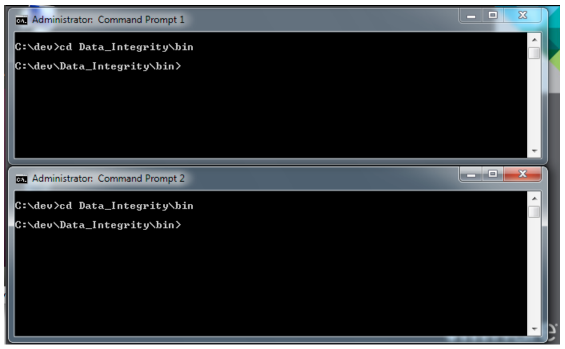
In this scenario we will pull the plug on one of the GemFire data nodes and continue updating some of our data. Our downed member will have out of date information, which would be catastrophic to our data integrity if this non-operational data were to be reintroduced.

We will restart the entire GemFire data fabric as a way to stress this capability. If we only restart the downed member the remaining members would populate it on the rejoin, the same as adding new capacity to your data fabric.

Step 1 – Rejoin: CD to bin directory.

For this scenario we are going to need two command prompts. Go ahead and open two command prompts and `cd` to the example `bin` directory.

Windows: `cd <path to example>\bin`
Linux: `cd <path to example>/bin`


Administrator: Command Prompt 1

```
C:\dev>cd Data_Integrity\bin
C:\dev\Data_Integrity\bin>
```

Administrator: Command Prompt 2

```
C:\dev>cd Data_Integrity\bin
C:\dev\Data_Integrity\bin>
```

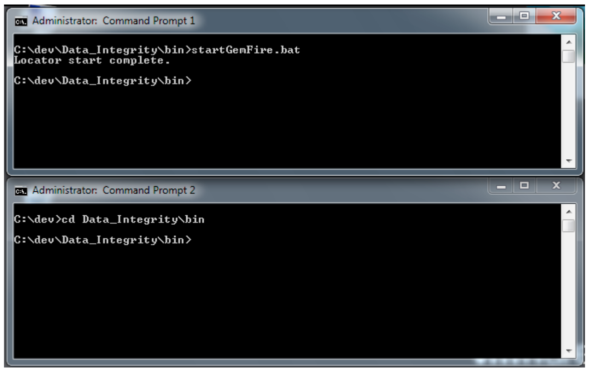
Figure 8 CD to the bin directory

Step 2 - Rejoin: Startup GemFire

We are going to start up the locator and three data management servers. The locator allows the servers to register themselves so they are discoverable by other servers and clients. The locator enables servers to partition or replicate the data based on the rules setup in the cache.xml.

For our example we have setup a redundancy of one. This means that the there is at least one redundant copy of the data spread across our distributed system. In the first command prompt window enter the following:

Windows: `startGemFire.bat`
Linux: `./startGemFire.sh`


Administrator: Command Prompt 1

```
C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>
```

Administrator: Command Prompt 2

```
C:\dev>cd Data_Integrity\bin

C:\dev\Data_Integrity\bin>
```

Figure 9 Starting up GemFire

Congratulations you have setup GemFire: a fault tolerant, highly available and low latency distributed data fabric. Best practices say that you wouldn't want a production system running on a single piece of hardware if high availability were a concern. We are running on a single piece of hardware for the sake of learning the basics.

This brings up an important deployment architecture issue. In a virtualized infrastructure it is possible for all of your virtual machines to migrate to the same host. If this happens you will have lost your hardware-based redundancy. This is something to consider when architecting your deployment strategy, especially when deploying on a virtualized infrastructure.

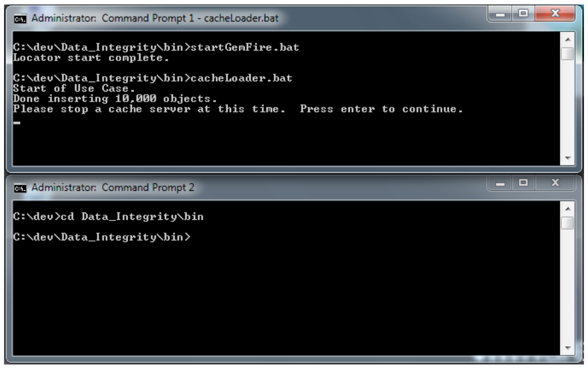
Step 3 - Rejoin: Load Data into GemFire

The client is going to contact the locator to find the available servers and start putting data into GemFire. Since we are attaching to a partitioned region, the client will start connecting to all of the servers once when we start inserting data. This is due to single hop optimization, which is turned on by default.

Once the connection is established our client code starts inserting 10,000 basic `SampleData` objects into our portioned region.

In the first Command Prompt window go ahead and run the above code.

Windows: `cacheLoader.bat`
Linux: `./cacheLoader.sh`



Administrator: Command Prompt 1 - cacheLoader.bat

```
C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>cacheLoader.bat
Start of Use Case.
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.
_
```

Administrator: Command Prompt 2

```
C:\dev>cd Data_Integrity\bin

C:\dev\Data_Integrity\bin>
```

Figure 10 After running 10,000 objects with the cacheLoader

So we have just inserted 10,000 object into the cache, each initialized with a value of 1. Now the `cacheLoader` will sit idle waiting for step 5.

Step 4 - Rejoin: Pull the Plug on a Server

It is time to stop one of servers.

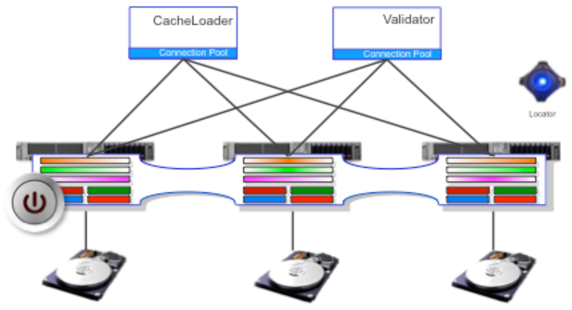


Figure 11 We are going to pull the plug on server 1

There is a lot that is going to happen with GemFire when we pull the plug on our server. Our two remaining servers in our fabric are setup to instantly rebalance their contents to attempt to maintain the redundancy of 1. Since we have 2 servers left, each server will have a copy of each other's data.

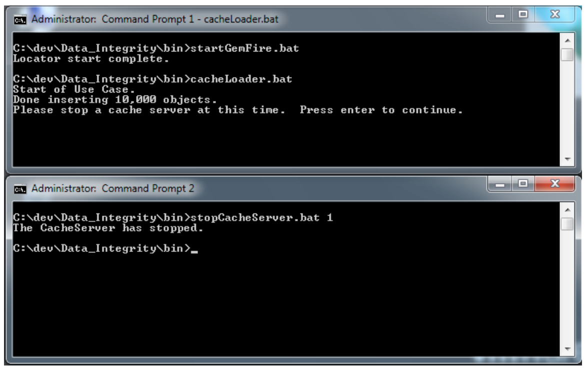
The time to rebalance is configurable with GemFire. A good example of waiting to rebalance is after rolling out a system patch. By waiting to rebalance you can prevent unnecessary work being done while the patched system comes back online.

Any clients connected to server 1 will automatically failover to one of the remaining servers.

Since we have persistence turned on our downed server will become inconsistent if there are any further updates to the data fabric. This server will instantly become a threat to our data consistency.

In command prompt 2 lets go ahead and stop server number 1.

Windows: `stopCacheServer.bat 1`
Linux: `./stopCacheServer.sh 1`



```
Administrator: Command Prompt 1 - cacheLoader.bat
C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>cacheLoader.bat
Start of Use Case.
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.
```

```
Administrator: Command Prompt 2
C:\dev\Data_Integrity\bin>stopCacheServer.bat 1
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>_
```

Figure 12 Stopped one if the cache servers

NOTE: The `stopCacheServer` script "nicely" shuts down a server using a GemFire command. If you would like to kill a server more abruptly, GemFire writes the PID of server in a log file. The log file can be found in: `<path to example>\data\server1\cacheserver.log`.

Look for something like this near the top:

```
Process ID: 5888
```

Then take the Process ID from the log file and kill it:

Windows: `tskill 5888`
Linux: `kill -9 5888`

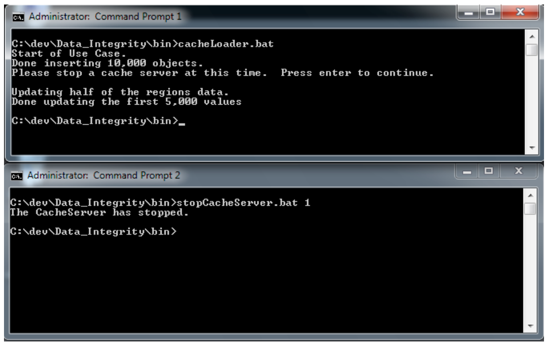
Step 5 - Rejoin: Create A Difference Between Online and Offline Servers

In this next step we are going to loop over half of our objects in our region and update `SampleData` object to contain the value two. By doing this, we are forcing the server that is currently powered off to have data that is inconsistent with the rest of the system. This will also yield an easily identifiable pattern to ensure that our data integrity has been maintained.

The pattern that we will be looking for later on is: `SampleData` with keys 0-4999 will have the value of two and `SampleData` with keys 5000-9999 will have the value of one.

Press enter to update the online servers to create the difference between servers.

Windows: `<enter>`
Linux: `<enter>`



```
Administrator: Command Prompt 1
C:\dev\Data_Integrity\bin>cacheLoader.bat
Start of Use Case.
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.

Updating half of the regions data.
Done updating the first 5,000 values

C:\dev\Data_Integrity\bin>_
```

```
Administrator: Command Prompt 2
C:\dev\Data_Integrity\bin>stopCacheServer.bat 1
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>
```

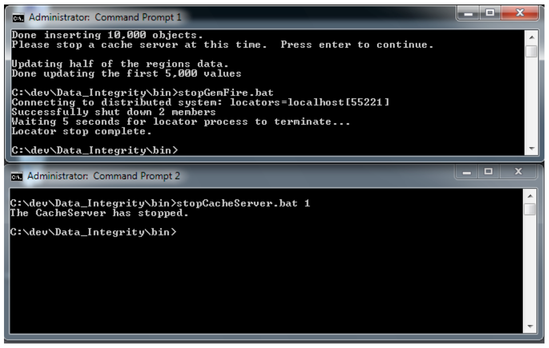
Figure 13 After updating half of our data.

Step 6 - Rejoin: Stop GemFire

At this point servers two and three contain valid data, and server one contains out of date information. If we were to introduce that data to our fabric the integrity would be compromised. However GemFire knows about all of the servers and their condition and meta data to safeguard your data. In order to stress test this ability we are going to shutdown the whole data fabric and restart it as a whole.

In Command Prompt 1 enter the following to shutdown GemFire:

Windows: `stopGemFire.bat`
Linux: `./stopGemFire.sh`



```
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.

Updating half of the regions data.
Done updating the first 5,000 values

C:\dev\Data_Integrity\bin>stopGemFire.bat
Connecting to distributed system: locators=localhost[55221]
Successfully shut down 2 members
Waiting 5 seconds for locator process to terminate...
Locator stop complete.

C:\dev\Data_Integrity\bin>
```

```
C:\dev\Data_Integrity\bin>stopCacheServer.bat 1
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>
```

Figure 14 Stopping the GemFire Data fabric.

Step 7 - Rejoin: Restart GemFire

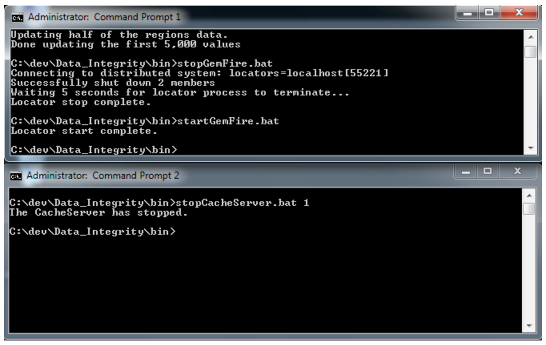
Now we are going to bring up all the servers that were in the original data fabric. As we bring up the data fabric it will actually wait for all the servers to rejoin. If a server cannot rejoin due to some issue a system administrator can revoke that missing members privileges from the data fabric. We will go into how to revoke a system in the next scenario, when we corrupt a server's disk store.

Once the systems have rejoined they go through a vetting process. This process is key to how GemFire maintains the integrity of the data stored within the data fabric. During this process server one will be discovered as out of date and its data is discarded.

After your data integrity is assured GemFire will rebalance the partition load amongst all of the servers that are currently participating in the data fabric. This rebalancing process is done while the system is operational and is a transparent operation to the systems using the data fabric.

In command prompt 1 restart the GemFire data fabric:

Windows: `startGemFire.bat`
Linux: `./startGemFire.sh`



```
Updating half of the regions data.
Done updating the first 5,000 values

C:\dev\Data_Integrity\bin>stopGemFire.bat
Connecting to distributed system: locators=localhost[55221]
Successfully shut down 2 members
Waiting 5 seconds for locator process to terminate...
Locator stop complete.

C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>
```

```
C:\dev\Data_Integrity\bin>stopCacheServer.bat 1
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>
```

Figure 15 Start up the GemFire Data Fabric.

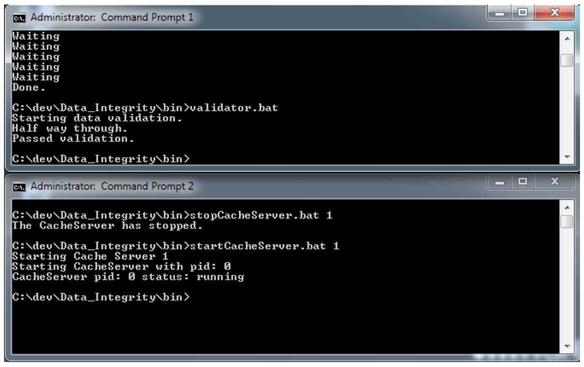
Step 8 - Rejoin: Validate

Now it is time to validate that everything worked as planned. The validator will loop through all of the keys and verify that the object values are what they should be.

There are two parts of the validation. To insure all the key-value pairs that we put into the cache exist and that the values of those objects are what they were designed to be. Our data pattern is keys 0-4,999 will have `SampleData` set to value of two. Keys 5,000-9,999 will have `SampleData` with a value of one. If either condition is not satisfied then the validator will output the reason and abruptly exit.

Lets go ahead and run the validation code, in command prompt one enter the following:

Windows: `validator.bat`
Linux: `./validator.sh`


Administrator: Command Prompt 1

```
Waiting
Waiting
Waiting
Waiting
Waiting
Done.

C:\dev\Data_Integrity\bin>validator.bat
Starting data validation.
Half way through.
Passed validation.

C:\dev\Data_Integrity\bin>
```

Administrator: Command Prompt 2

```
C:\dev\Data_Integrity\bin>stopCacheServer.bat 1
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>startCacheServer.bat 1
Starting Cache Server 1
Starting CacheServer with pid: 0
CacheServer pid: 0 status: running

C:\dev\Data_Integrity\bin>
```

Figure 16 Passed Validation

We have shown how GemFire can survive an impromptu server shutdown transparent to your application. Your data's integrity is safety maintained in the remaining servers within the GemFire data fabric. Then we restarted the GemFire data fabric and it was able to detect that server 1 held stale information and then invalidated its data.

Can you think about a harder scenario? If so give it a try. In the next section we will exploring what happens when the data becomes corrupt on one of the servers in the GemFire data fabric.

If you want to try out the next scenario you can leave GemFire running. If you wish to stop at this point you can gracefully shut down GemFire by running the following in a command prompt:

Windows: `stopGemFire.bat`
Linux: `./stopGemFire.sh`

Data Corruption Scenario

A hardware or software failure in your data center is something that can cause serious imprecisions to any business. Lets model corrupted data on our storage system. This would be a catastrophic event if it were to happen. However GemFire is mitigating this risk by maintaining redundant copies of your data on multiple hardware and storage systems.

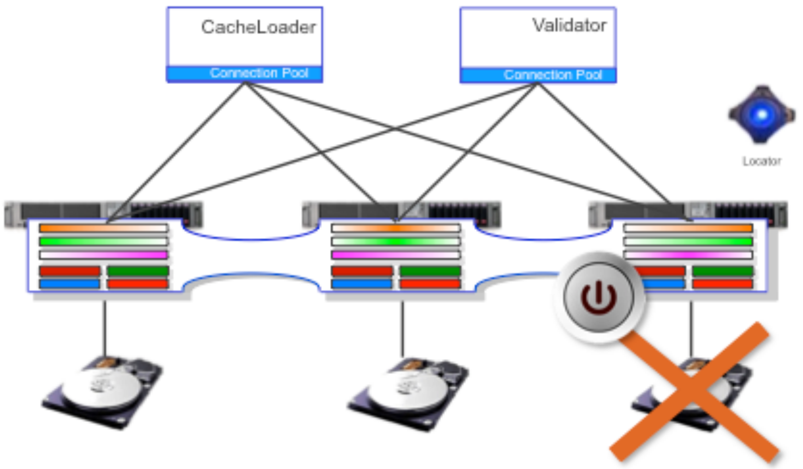


Figure 17 Toggle the power and corrupt the server files.

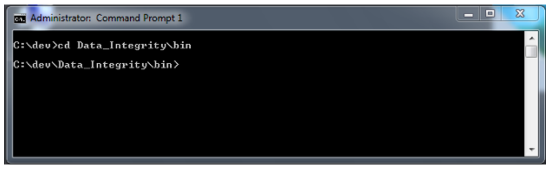
If you already have a valid data fabric and just finished up the validation step in the last scenario you can skip to step 5.

In this example we only need one command prompt.

Step 1 - Data Corruption: CD to bin

Change directory to the example bin directory:

Windows: `cd <path to example>\bin`
Linux: `cd <path to example>/bin`



```
C:\dev>cd Data_Integrity\bin
C:\dev\Data_Integrity\bin>
```

Figure 18 CD to the bin directory

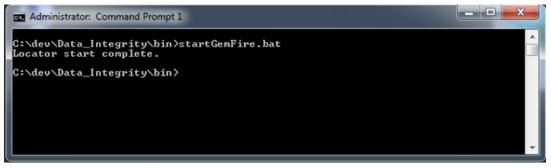
Step 2 - Data Corruption: Startup GemFire

We are going to start up the locator and three data management servers. The locator allows the servers to register themselves so they are discoverable by other servers and clients. This enables servers to partition or replicate the data based on the rules setup in the cache.xml.

For our example we have setup a redundancy of one. This means that the there is at least one redundant copy of the data spread across our distributed system. In the summary we will discuss how to choose a redundancy that makes sense for your deployment of GemFire.

Start the GemFire data fabric.

Windows: `startGemFire.bat`
Linux: `./startGemFire.sh`



```
C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>
```

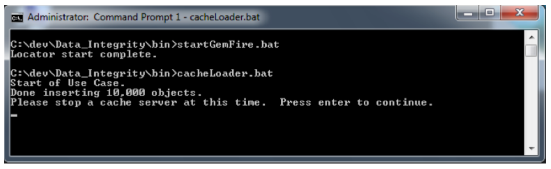
Figure 19 Start the GemFire data fabric.

Step 3 - Data Corruption: Load Data into GemFire

Here the client is going to contact the locator to find the available servers and start putting data into GemFire. Since we are attaching to a partitioned region, the client will start connecting to all of the servers once when we start inserting data. This is due to single hop optimization, which is turned on by default.

Once the connection is established our client code starts inserting 10,000 basic `SampleData` objects into our portioned region. Launch the `cacheLoader`
to establish a baseline cache that can be verified.

Windows: `cacheLoader.bat`
Linux: `./cacheLoader.sh`



```
C:\dev\Data_Integrity\bin>startGemFire.bat
Locator start complete.

C:\dev\Data_Integrity\bin>cacheLoader.bat
Start of Use Case.
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.
_
```

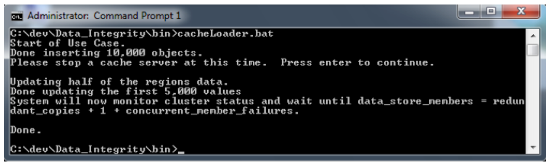
Figure 20 Halfway through establishing our baseline.

Step 4 - Data Corruption: Finish Loading Data into GemFire

Here is where we are going to change things from the rejoin scenario. Since we have already shown that GemFire can handle when servers are inconsistent. There is no need to have our `cacheLoader` wait or shutdown a server to cause an inconsistency.

Lets just let the `cacheLoader` continue updating the intact data fabric without pulling the plug on any of the servers.

Windows: `<enter>`
Linux: `<enter>`



```
C:\dev\Data_Integrity\bin>cacheLoader.bat
Start of Use Case.
Done inserting 10,000 objects.
Please stop a cache server at this time.  Press enter to continue.

Updating half of the regions data.
Done updating the first 5,000 values
System will now monitor cluster status and wait until data_store_members = redun
dant_copies + 1 + concurrent_member_failures.

Done.

C:\dev\Data_Integrity\bin>_
```

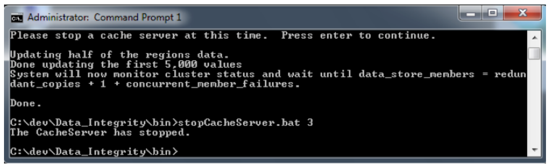
Figure 21 After pressing enter on the cacheLoader without killing a server

Step 5 - Data Corruption: Stop a GemFire Server to Corrupt

At this point we have a verifiable baseline stored in our data fabric. Now we can model a data failure incident on one of our servers. For this we will choose server 3 to be the target.

Pull the plug on server 3:

Windows: `stopCacheServer.bat 3`
Linux: `./stopCacheServer.sh 3`



```
Please stop a cache server at this time.  Press enter to continue.

Updating half of the regions data.
Done updating the first 5,000 values
System will now monitor cluster status and wait until data_store_members = redun
dant_copies + 1 + concurrent_member_failures.

Done.

C:\dev\Data_Integrity\bin>stopCacheServer.bat 3
The CacheServer has stopped.

C:\dev\Data_Integrity\bin>
```

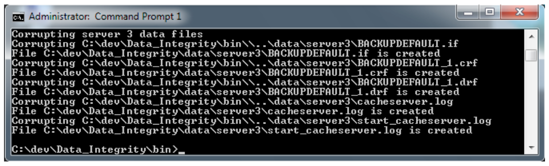
Figure 22 Pulling the plug on server 3

Step 6 - Data Corruption: Corrupt GemFire data files.

In this step we will corrupt the GemFire data files beyond repair. To do this we will loop through all of the key files for a given server setting them to garbage. This would model a complete failure that would bring a system down for hours if not days.

Corrupt the server files:

Windows: `corruptData.bat 3`
Linux: `./corruptData.sh 3`



```
Corrupting server 3 data files
Corrupting C:\dev\Data_Integrity\bin\\..\data\server3\BACKUPDEFAULT.if
File C:\dev\Data_Integrity\data\server3\BACKUPDEFAULT.if is created
Corrupting C:\dev\Data_Integrity\bin\\..\data\server3\BACKUPDEFAULT_1.crf
File C:\dev\Data_Integrity\data\server3\BACKUPDEFAULT_1.crf is created
Corrupting C:\dev\Data_Integrity\bin\\..\data\server3\BACKUPDEFAULT_1.drf
File C:\dev\Data_Integrity\data\server3\BACKUPDEFAULT_1.drf is created
Corrupting C:\dev\Data_Integrity\bin\\..\data\server3\cacheserver.log
File C:\dev\Data_Integrity\data\server3\cacheserver.log is created
Corrupting C:\dev\Data_Integrity\bin\\..\data\server3\start_cacheserver.log
File C:\dev\Data_Integrity\data\server3\start_cacheserver.log is created

C:\dev\Data_Integrity\bin>_
```

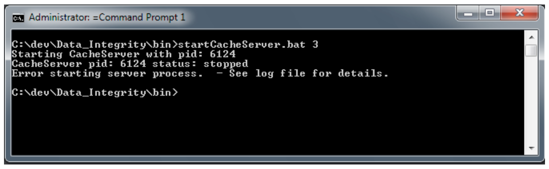
Figure 23 After corrupting server 3 data files

Step 7 - Data Corruption: Start Corrupted Server

Now that the server files are corrupted, lets see what happens when we try to start up GemFire.

Start GemFire server 3:

Windows: `startCacheServer.bat 3`
Linux: `./startCacheServer.sh 3`



```
C:\dev\Data_Integrity\bin>startCacheServer.bat 3
Starting CacheServer with pid: 6124
CacheServer pid: 6124 status: stopped
Error starting server process.  - See log file for details.

C:\dev\Data_Integrity\bin>
```

Figure 24 GemFire didn’t start the server with the corrupted data files.

Step 8 - Data Corruption: Revoking the Corrupted Disk Store

When the server is in a highly corrupted state like we have here, GemFire will prevent it from coming back online.   This is a good thing because a system administrator can get involved and trouble-shoot what is going on before an unreliable system can be brought back on-line.

Since our data integrity is still protected by the remaining data fabric, the system administrator has many courses of action. They can perform a hot swap of new hardware to something less severe like running hardware and software diagnostics. We are going to model a case where we want to turn the existing member back online.

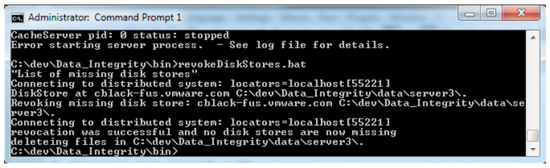
If the system administrator has either swapped in new hardware or certified the existing hardware, they have to revoke server 3’s disk store from the fabric. By revoking the disk store, we have detached the disk store from the fabric.

Some caution must now be placed on this now detached disk store. If we let it come back on-line by itself it will consider itself a primary data holder. Best practice dictates a new/revoked member join an already running data fabric.

In this step we will query the data fabric to see if there are any missing disk stores.   We will then revoke any missing disk stores from the fabric. Then we will remove the corrupted files from disk. In other words we are removing the knowledge that server 3 was part of the data fabric and cleaning up its corrupt data files – bring it back to a “new” state.

Revoke the server 3 disk store:

Windows: `revokeDiskStores.bat`
Linux: `./revokeDiskStores.sh`



```
CacheServer pid: 0 status: stopped
Error starting server process.  - See log file for details.

C:\dev\Data_Integrity\bin>revokeDiskStores.bat
"List of missing disk stores"
Connecting to distributed system: locators=localhost[55221]
DiskStore at cblack-fus.vmware.com C:\dev\Data_Integrity\data\server3\.
Revoking missing disk store: cblack-fus.vmware.com C:\dev\Data_Integrity\data\se
rver3\.
Connecting to distributed system: locators=localhost[55221]
revocation was successful and no disk stores are now missing
deleteing files in C:\dev\Data_Integrity\data\server3\.
C:\dev\Data_Integrity\bin>
```

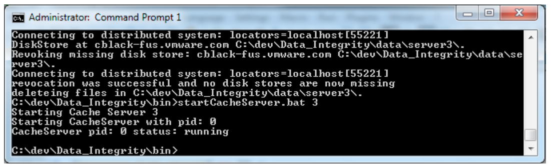
Figure 25 After successfully revoking the disk store.

Step 8 - Data Corruption: Startup the Newly Revoked Server

After revoking the disk store and cleaning up the corrupt files, we are now able to bring the system back on-line. Keeping with best practice with the other servers already up and online. This will cause this server to join the fabric and be initialized with the data already in the fabric. In essence this is a new server joining the data fabric.

Bring the downed server back online:

Windows: `startCacheServer.bat 3`
Linux: `./startCacheServer.sh 3`



```
Connecting to distributed system: locators=localhost[55221]
DiskStore at cblack-fus.vmware.com C:\dev\Data_Integrity\data\server3\.
Revoking missing disk store: cblack-fus.vmware.com C:\dev\Data_Integrity\data\se
rver3\.
Connecting to distributed system: locators=localhost[55221]
revocation was successful and no disk stores are now missing
deleteing files in C:\dev\Data_Integrity\data\server3\.
C:\dev\Data_Integrity\bin>startCacheServer.bat 3
Starting Cache Server 3
Starting CacheServer with pid: 0
CacheServer pid: 0 status: running

C:\dev\Data_Integrity\bin>
```

Figure 26 Cache successfully rejoined the distributed fabric.

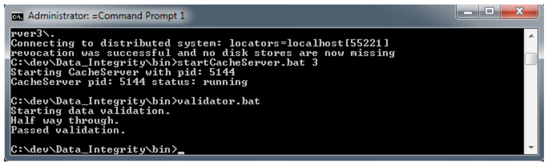
Step 9 - Data Corruption: Validate the Cache

With our data fabric configuration, our just launched server automatically starts sharing the load. Redundant copies of our data will be shared among all the members of the data fabric, ensuring that your data's integrity remains intact incase of another extreme failure.

With the server up and running the only thing to do is validate the integrity of the cache.

Validate the cache:

Windows: `validator.bat`
Linux: `./validator.sh`



```
rver3\.
Connecting to distributed system: locators=localhost[55221]
revocation was successful and no disk stores are now missing
C:\dev\Data_Integrity\bin>startCacheServer.bat 3
Starting CacheServer with pid: 5144
CacheServer pid: 5144 status: running

C:\dev\Data_Integrity\bin>validator.bat
Starting data validation.
Half way through.
Passed validation.

C:\dev\Data_Integrity\bin>_
```

Figure 27 Cache integrity validated.

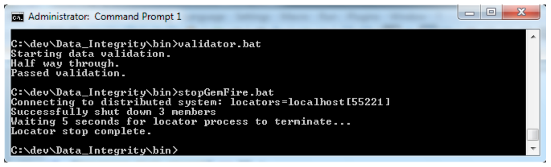
At this point the validator will have checked to see if we maintained data integrity of the data fabric.

Just like in real life, no one can reliably know when hardware or software will fail. However, deployed properly GemFire is designed to help mitigate catastrophic data integrity issues. GemFire was able to continue to operate while one of its servers was offline getting serviced.

Step 10 - Data Corruption: Stop GemFire

Gracefully shutdown the GemFire distributed fabric.

Windows: `stopGemFre.bat`
Linux: `./stopGemFire.sh`



```

C:\dev\Data_Integrity\bin>validator.bat
Starting data validation.
Half way through.
Passed validation.

C:\dev\Data_Integrity\bin>stopGemFire.bat
Connecting to distributed system: locators=localhost[55221]
Successfully shut down 3 members
Waiting 5 seconds for locator process to terminate...
Locator stop complete.

C:\dev\Data_Integrity\bin>
```

Figure 28 Shutdown of the distributed fabric.

Summary

We have shown that GemFire can automatic handle members entering and exiting the distributed system. Also, we have shown how GemFire alerts your system administrators when there is a hardware or software integrity issue that needs their attention. GemFire kept the integrity of the data intact and continued to be operational regardless of the machines exiting or joining the data fabric.

In these examples, our server data redundancy was set to 1. This means there are at least 2 copies of your data spread out across the servers. You can alter the redundancy based on your risk tolerance from 0 to 3.

To calculate the number of members you need for redundant data storage, assuming that all your members are on separate machines, use this formula as a general guideline:

data_store_members = redundant_copies + 1 + concurrent_member_failures

Where:

- data_store_members = GemFire Servers
- redundant_copies = Redundant copies of your data
- concurrent_member_failures = Number of servers that can fail at the sametime.

Use the calculation as the beginning point in your planning process, not the final answer. This formula doesn't always work. Where redundant_copies is less than concurrent_member_failures, for example, no number of members can guarantee that you won't have data loss.

GemFire gives you:

High Scalability

- Scalability is achieved through dynamic partitioning of data across many member nodes and spreading the data load across the servers.
- For 'hot' data, the system can be dynamically expanded to have more copies of the data.
- Application behavior can also be provisioned and routed to run in a distributed manner in proximity to the data it depends on.

Continuous Availability

- In addition to guaranteed consistent copies of data in memory across servers and nodes, applications can synchronously or asynchronously persist the data to disk on one or more nodes.
- GemFire's shared-nothing disk architecture ensures very high levels of data availability.