Bridging the Cloud with In-Memory Data Grid

September 2012

A solution executed by:

&

razorfish™

Bridging the Cloud with In-Memory Data Grid

It's hard to ignore the buzz around cloud computing, given all the exciting innovation happening in this space. Even the most conservative IT organizations have now been forced to consider using one of the many available cloud platforms to improve on speed of delivery, cost saving, and reliability.

One of the most attractive features of today's cloud offerings is that they enable IT to extend the capacity of their solutions beyond the scope of on-premise servers. This can be in terms of high availability, disaster recovery, or scaling to meet planned and unplanned spikes in usage.

A major challenge in moving applications from on-premise datacenters to public clouds is the reluctance to store sensitive data on the cloud, for various reasons: perceived lack of control over the storage, security concerns or non-compliance issues when data is stored beyond the enterprise's boundaries, or the need to store the data on-premise for other internal applications to access. There might also be cases where the data resides within systems or servers that simply have no equivalent component available on the cloud, such as a proprietary data store like a file system, or mainframe database.

A possible approach in such a scenario is to use a hybrid architecture where the bulk of the customer-facing application is moved to the public cloud, allowing it to leverage all the cloud's advantages (scalability, cost, redundancy, DR, etc.), while the data store continues to reside on-premise. The application on the cloud must be able to access this data when needed. This allays some of the concerns about storing data outside the enterprise and might be less intrusive and in line with strategic enterprise decisions. Because this approach frees up data from legacy slow-changing technologies, it might also offer up new use cases for the data, and it is also easier to sell internally.

Solving the Problem of Data Access from the Cloud

There are many strategies for accessing on-premise data from the cloud. Two of the most common are exposing the data using web-services, and replicating the data to the cloud. Both are briefly described below. This article also describes an alternative that uses an In-Memory Data Grid (IMDG) as a solution.

Exposing Data Using Web-Services

The simplest and most common approach is to create a service facade layer above the data, and then have the cloud/remote applications access the data through this service layer. The service is usually accessible externally through a REST or SOAP endpoint. For sensitive data, SSL can be used as the transport for these service endpoints.

This approach meets the need to store data on-premise, while also providing a service layer abstraction for data. It fits nicely with industry best practices for setting up with SOA. Any necessary governance and security controls can be applied to this service layer.

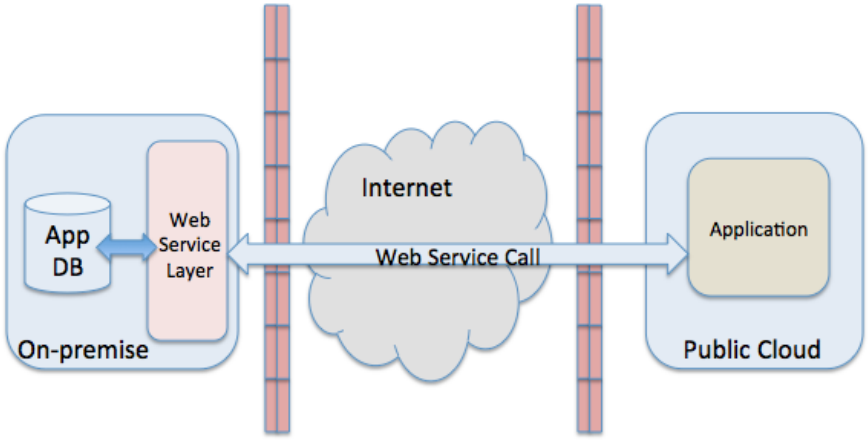

Fig. 1. Data Accessed via web services

While this model works in many cases, there are some drawbacks to be considered. Each user request has to go through an additional network hop before it can be satisfied: as a request comes from user to cloud server, the cloud server must request data from the on-premise web-service, increasing the response latency. Also, if the data web service layer is very granular, a single user request can result in multiple data web service calls.

Some drawbacks can be mitigated by designing the application so that the data web service requests are made by the user client browser, and then composing the UI based on the request, or by using service bus orchestration for web service calls to be combined together, etc. In any case, these approaches add complexity to the application.

Creating a Replica of the Database in the Cloud

Another approach is to replicate the data from the on-premise data store to a cloud data store. This solves the problem of adding application complexity, but it does not address the original concern of not storing sensitive data on a public cloud. It also introduces the complexity of having to synchronize the data and coming up with an appropriate replication strategy. Replication strategies are always non-trivial exercises, depending on how many other applications are updating and accessing this data.

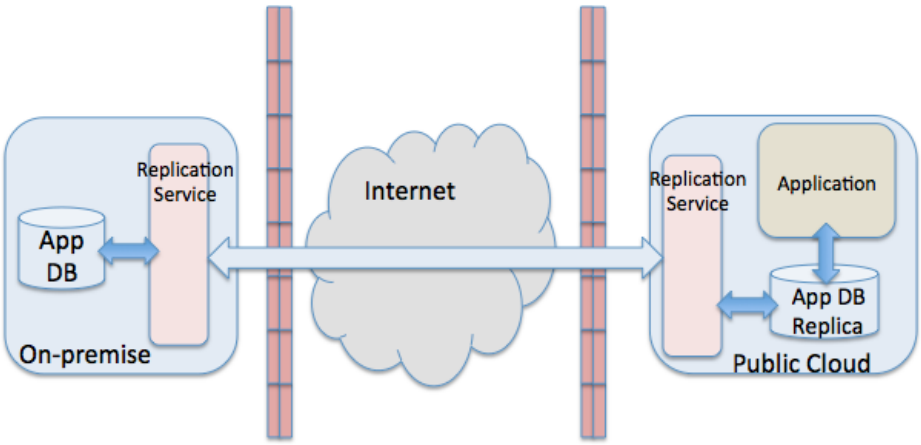

Fig. 2. Data replicated at the public cloud

Making Data Available on the Cloud

Another approach to keep the complexity of the application from increasing is to make the data from the on-premise stores available to the user-facing applications on the cloud. In this case, the data is not stored on the cloud but is made available for processing in the cloud-layer via in-memory replication. Any changes made to the data are reflected back upon the on-premise data store in near real time.

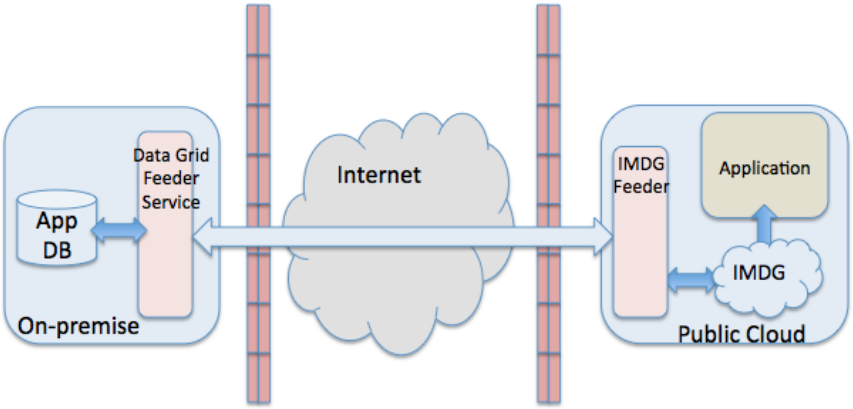

Fig. 3. Data available on IMDG in public cloud

In-memory replication enables replication in milliseconds rather than a typical delay of minutes for disk replication solutions. Also, when compared to the web-service based approach, this approach eliminates the unnecessary network hops needed to service user requests, thereby improving end user performance. This approach brings data to the cloud, where the application can interact with it, and the application is completely shielded from the complexity of having to persist or replicate data back to the on-premise store. The use of an IMDG also means that while the data is available on the cloud, it is only available in memory and is never stored on a disk in the cloud.

Comparing the Three Approaches

	Web Service	DB Replication	IMDG
Latency	✓ **WAN Latency** Data is retrieved from on-premise via a web Service	✓✓ **Disk Latency** Data is retrieved from database running in Cloud	✓✓✓ **In Memory Latency** Data is retrieved from IMDG running in public cloud
Failure of On-Premise Database	✗ **Application unavailable**	✓ **Application still available**	✓ **Application still available**
Failure of On-Premise Data Center	✗ **Application unavailable**	✓ **Application still available**	✓ **Application still available**
Failure Of Network Connectivity	✗ **Application unavailable**	✓ **Application still available**	✓ **Application still available**
Bi-Directional Updates	**N/A** Data is available only in on-premise database	✗ **Conflicts are possible. Not easy to resolve conflicts.**	✓ **Bi-directionality with conflict resolution**
Security	✓ **Data is Secure** Data is available only in on-premise database	✗ **Data is in permanent cloud store** Data is stored in a cloud DB	✓ **Data is secure** Data is stored only in memory on cloud
Effort to Migrate Existing Application	**High** Application needs to be modified to interact with web services.	**Low** Application uses DB running in cloud -- should be a simple switch	**Medium** Application interacts with IMDG, not DB. If the app already uses JPA it's an easy switch

The rest of this document describes this approach of making on-premise data available on the cloud using IMDG

Leveraging the IMDG to Bring Data to the Cloud

IMDG Overview

An IMDG is a distributed non-relational data or object store. It can be distributed to span more than one server. IMDGs usually support linear scaling to support high loads, data partitioning, redundancy, and automatic data recovery in case of failures. Most IMDGs also support multimode topologies that span WANs.

In a solution using an IMDG, data from the on-premise data store is loaded into a node in the IMDG cluster. The data grid is set up to replicate data from this node, across the WAN, to one or more IMDG nodes that run on a public cloud server. Applications running on the public cloud server can access and update data from any of the IMDG nodes on the public cloud. Any updates made to the IMDG are played back to other nodes in the cluster, including the node running on the on-premise data center. These updates can then be persisted to the on-premise store.

GigaSpaces XAP was used to implement this approach, as described below.

GigaSpaces XAP Overview

GigaSpaces XAP is a Java-based platform that enables us to set up a ‘space’-based architecture where you can host both data and business logic. The XAP platform provides scalability, reliability, and consistency while supporting high throughput and low latency. The XAP application platform supports clustering, partitioning, messaging, persistence, synchronous or asynchronous processing, transaction support and many more features.

At the core of this approach is the concept of a 'space', service beans, and processing units. A 'space' is a logical unit that holds data objects (think POJOs) in memory. A space can be configured to have one or more backups. Service beans are application logic components that can access and update data objects from a space. Spaces and service beans are bundled together as ‘processing units' (PUs) to be deployed on the XAP platform. A processing unit could also be made up of a web application (e.g. a WAR). A data grid is a set of space instances, running within their respective PU instances.

For more details: http://www.gigaspaces.com/wiki/display/XAP9/9.0+Documentation+Home

Bringing the Data to the Cloud

Building on the concept of spaces, the idea is to set up a ‘space’ within the on-premise environment that contains the necessary data, and replicate this space to cloud, making the data available to components running on cloud servers.

This scenario includes two servers, each running XAP. One server (called the local server) is an on-premise server, while the other (called the cloud server) runs on the public cloud (e.g. AWS EC2). The local server loads the data from the on-premise database into a GigaSpaces XAP space cluster. The space on the local server is replicated to a space on the cloud server. The cloud server also hosts processing

units that contain both service beans and web applications. Any data that is updated on the cloud server space is persisted asynchronously to the on-premise database via local server space.

GigaSpaces XAP includes out-of-the-box components to load data into spaces, replicate spaces across WANs, and persist data from a space back to the data store.

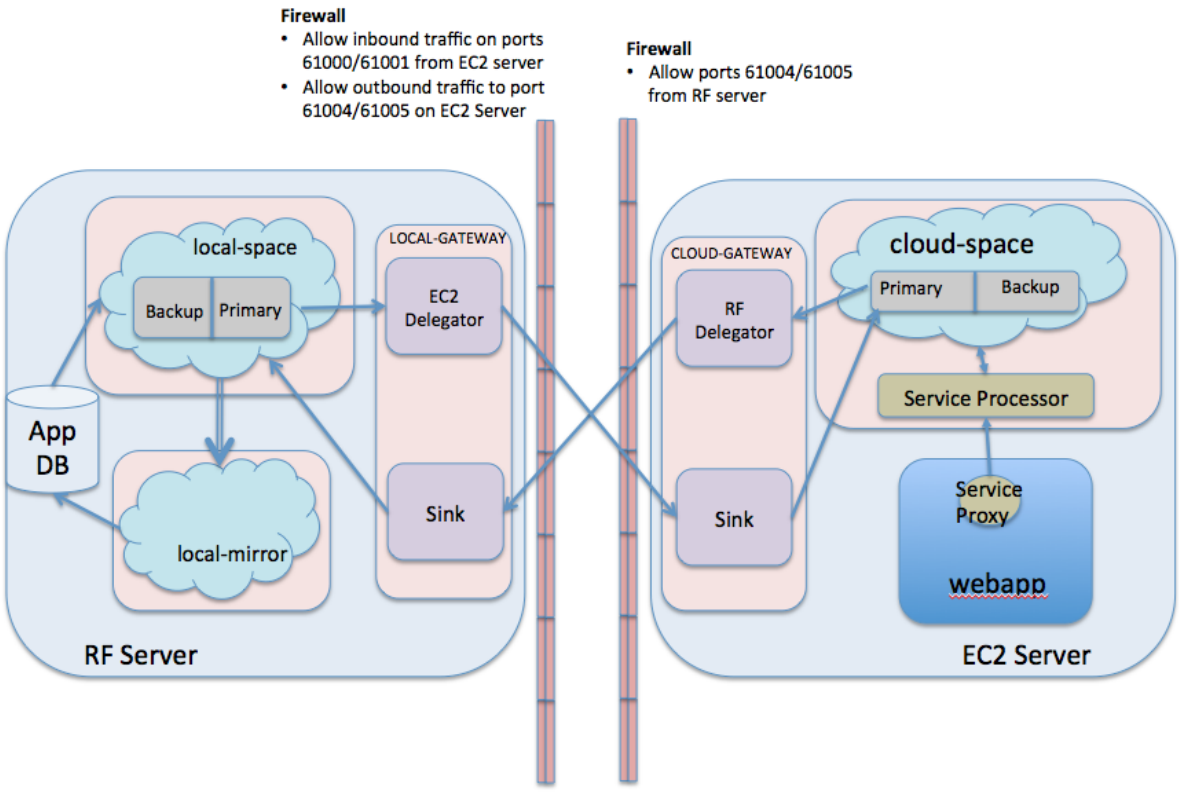

Fig. 4. Proposed Architecture using GigaSpaces IMDG

Populating a Space with Data from a Database

GigaSpaces XAP's External Data Source (EDS) component can be used to load data from a database to a space. We use the built-in Hibernate EDS implementation that pre-loads the data from the database using Hibernate. If Hibernate is not an option, one could also write a custom EDS that implements the DataProvider and/or DataPersister interfaces to suit custom requirements.

The EDS includes a mechanism to pre-populate the space with data before it is available for client access. This mechanism used to pre-load the data into the XAP data grid is called Initial Load. The Hibernate EDS Initial Load implementation is made aware of the Objects to be loaded via configuration and simple annotations. In the example below, EDS will populate the space with 'Person' objects.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
                    ...
                    ...
 </bean>

 <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
```

```
  <property name="dataSource" ref="dataSource"/>
  <property name="annotatedClasses">
   <list>
    <value>com.rfpoc.app.model.Person</value>
   </list>
  </property>
                    …
                    …
 </bean>

 <bean id="hibernateDataSource"
class="org.openspaces.persistency.hibernate.DefaultHibernateExternalDataSource">
  <property name="sessionFactory" ref="sessionFactory"/>
 </bean>
       …
       …

 <os-core:space id="local-space" url="/./local-space" schema="persistent"
     external-data-source="hibernateDataSource">
                                …
                                …
</os-core:space>
```

The Person Class will be defined as below. The table layout is inferred from the JPA annotations.

```
 @Entity
 @Table(name="USER_TABLE")
 @SpaceClass
 public class Person implements Serializable{
  public Person(){
  }

  @Id
  @Column(name="ID")
  @SpaceId(autoGenerate=false)
  @SpaceRouting
  public Integer getId() {
   return id;
  }
  public void setId(Integer id) {
   this.id = id;
  }
  @Column(name="FIRST_NAME")
  public String getFirstName() {
   return firstName;
  }
  public void setFirstName(String firstName) {
   this.firstName = firstName;
  }
 …
 …
 } // end Person class
```

Mirror the Space with Asynchronous Persistency

XAP's Mirror Service provides asynchronous persistence of data to the DB. As the name indicates, a mirror is a conduit for changes made to the source spaces and played back to the persistent store, in this case the on-premise database. All changes

to the source spaces are played back into the mirror transactionally, in the order they were made, providing data consistency. If the mirror is unavailable for any reason, the changes are queued up at the source spaces as redo-logs until they can be sent over to the mirror.

As above, the mirror processing unit is also configured with a DataSource, SessionFactory, and default Hibernate EDS. The main difference in this case is that the space configuration indicates that its schema type is 'mirror'. The data source is marked as read-write for the mirror space and read-only for the processor.

```
    <bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
                                    …
                                    …
     </bean>

     <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
       <property name="dataSource" ref="dataSource"/>
       <property name="annotatedClasses">
   <list>
    <value>com.rfpoc.app.model.Person</value>
   </list>
  </property>
                    …
                    …
 </bean>

 <bean id="hibernateDataSource"
class="org.openspaces.persistency.hibernate.DefaultHibernateExternalDataSource">
  <property name="sessionFactory" ref="sessionFactory"/>
 </bean>
          …
          …
 <os-core:space id="mirror" url="/./local-mirror " schema="mirror"
     external-data-source="hibernateDataSource">
                                    …
                                    …
 </os-core:space>
```

The Cloud Space

Just as we defined and instantiated a space on the local server, we will also instantiate a space on the cloud server. The cloud space does not need an EDS configured, as it is not connected to a database. We will use Executor Based Remoting, defining a remote service that contains the business logic to make updates to data in the cloud space. Event-based remoting can also be used to update space objects. Defining the service simply requires using the @RemotingService annotation and the "ServiceExporter" component to make all configured services available for remote invocation.

Gateway and Sink Endpoints for Remote and Local Spaces

As spaces are located in different physical locations, we leverage XAP's multisite replication component (also referred to as WAN Gateway) to replicate data state changes across the distributed topology. A space can replicate to another space via a Gateway delegator component. Each delegator component is configured to delegate

state changes to a Sink. Thus, changes in a space's state flow through the Gateway it is configured to use, and this delegator talks to the Sink component in the cloud space to dispatch the changes. To enable XAP to look up and discover Gateway components located on remote servers, we have to open up a pair of ports per gateway (one discovery port and one communication port) to allow the communication between the local and cloud server. This communication can also be configured to use SSL as the transport.

```
        .....
  <os-gateway:delegator id="delegator" local-gateway-name="RF-GATEWAY"
gateway-lookups="gatewayLookups">
    <os-gateway:delegation target="CLOUD-GATEWAY"/>
  </os-gateway:delegator>

  <os-gateway:sink id="sink" local-gateway-name="RF-GATEWAY"
gateway-lookups="gatewayLookups"
         local-space-url="jini://*/*/local-space">
    <os-gateway:sources>
      <os-gateway:source name="CLOUD-GATEWAY"/>
    </os-gateway:sources>
  </os-gateway:sink>
        ...

  <os-gateway:lookups id="gatewayLookups">
    <os-gateway:lookup gateway-name="CLOUD-GATEWAY" host="cloud-server"
discovery-port="61001" communication-port="61000"/>
    <os-gateway:lookup gateway-name="RF-GATEWAY" host="local-server"
discovery-port="61005" communication-port="61004"/>
  </os-gateway:lookups>
```

For more info: http://www.gigaspaces.com/wiki/display/XAP9/Multi-Site+Replication+over+the+WAN

Once the remote and local gateway connections have been looked-up and connection has been established, we do need to ensure that existing data is made available to the cloud space from the local space. This process is called Bootstrapping. Bootstrapping occurs via the Sink components.

For more info:
http://www.gigaspaces.com/wiki/display/XAP9/Replication+Gateway+Bootstrapping+Process

Note : Since this example was run on an AWS server with a public and private IP we had to set up the network configuration file to provide the GigaSpaces Lookup services the NAT mappings to ensure proper discovery of components. The network_mapping.config file on the on-premise GigaSpaces installation had the <private-ip>:<port> mapped to the <external-ip>:<port>

e.g.,
10.38.7.194:61004,109.32.251.116:61004
10.38.7.194:61005,109.32.251.116:61005

Web Applications with Remote Proxies

All that remains is for the web application to be deployed within a processing unit in the XAP container on the cloud server. XAP supports standard Java WAR artifacts using an embedded Jetty container. Since the web application is just another PU, it too can leverage all the XAP features like scalability, reliability, etc. The web application uses a service proxy to locate the Remote service and invokes the necessary business logic there by making changes to space objects as needed.

Conclusion

As seen above, making an existing web application use cloud infrastructure is a simple process. All that is needed is to:

- Load data to a local grid node (configure the External Data Source and the local space)
- Configure the remote grid node (configure the cloud space)
- Ensure replication between local and remote nodes (configure the local and cloud gateways)
- Firewall changes to allow communication between the local and cloud servers
- Extract the DB update logic to work with the data grid (use a remoting service component)
- Update the existing web application, typically the DAOs, to use a service proxy invocation

This example was executed with GigaSpaces XAP but the same approach can be used with any comparable IMDG provider. While this approach can be used to serve user requests from the public cloud alone, it can also be used to support a 'cloud bursting' deployment model to meet peak or overload capacity, or free up precious local resources for more business-critical applications. The relative simplicity of this approach makes it worth considering as a solution that can be a bridge to get the best of both private hosting and public cloud worlds.