Sensor Data Distribution With Robustness and Reliability – Toward Distributed Components Model

Charles Lee
SAIC
c/o NASA Ames Research Center
Moffett Field, CA 94035
650-604-6054
charles.lee@saic.com

Richard L. Alena
NASA Ames Research Center
Moffett Field, CA. 94035
650-604-0262
Richard.L.Alena@nasa.gov

Abstract—In planetary surface exploration mission, sensor data distribution is required in many aspects, for example, in navigation, scheduling, planning, monitoring, diagnostics, and automation of the field tasks. The challenge is to distribute such data in the robust and reliable way so that we can minimize the errors caused by miscalculations, and misjudgments that based on the error data input in the mission. The ad-hoc wireless network on planetary surface is not constantly connected because of the nature of the rough terrain and lack of permanent establishments on the surface. There are some disconnected moments that the computation nodes will re-associate with different repeaters or access points until connections are reestablished. Such a nature requires our sensor data distribution software robust and reliable with ability to tolerant disconnected moments. This paper presents a distributed components model as a framework to accomplish such tasks. The software is written in Java and utilized the available Java Message Services schema and the JBoss implementation. The results of field experimentations show that the model is very effective in completing the tasks.

1. INTRODUCTION

Sensor data is essential to many aspects of planetary surface exploration mission. For planning, the sensor data are needed as inputs of the location, environment, and distances. For scheduling, senor data are needed for calculation of the duration, time, position, and routs. For operation, sensor data needed to calculate the location, progress, the health status, etc. The data acquired from sensors are real time stream and are needed by real time. Parallel processing of those data to the end need introduces the reliability issue. The major causes of the less reliable are large communication overhead, and difficult of multiple threading programming. To distribute sensor data in the reliable way, we experimented different schema, framework to find out the robust and reliable system. We finally selected the Message oriented middleware as distributed infrostructure [3]. Message-oriented middleware is a category of inter-application communication software that presents an asynchronous message-passing model as opposed to a request/response model. As far as the

client's concern, it is close to real-time processing [4]. Most MOM systems are based around a message queuing system. The primary advantage of a message-oriented communications protocol is the ability to store, route, and resend a message that is to be delivered.

Most MOM systems provide a persistent storage to hold messages until they are successfully transferred. This means that it is not necessary for both the sender and receiver to be connected at the same time. This is useful for dealing with faulty connections, unreliable networks, and timed connections. It also means that if a receiver fails to receive a message for any reason, the sender can continue unaffected, since the messages will be held in the message store and will be transmitted when the receiver reconnects.

MOM systems present two messaging models:

- Point to point: This model [2] is based on message stores known as queues. A sender sends a message to a specified queue. A receiver receives messages from the queue. A queue can have multiple senders and receivers, but an individual message can only be delivered to one receiver. If multiple receivers are listening for messages on a queue, the underlying MOM system usually determines which receiver will receive the next message. If no receivers are listening on the queue, messages remain in the queue until a receiver attaches to the queue.
- Publish Subscribe This model [1] is based on message stores known as topics. Publishers send messages to a topic. Subscribers retrieve messages

from a topic. Unlike the point-to-point model, many subscribers can receive the same message.

A message-driven bean must declare deployment information about itself in a deployment-descriptor file named *ejb-jar-xml*. The EJB container handles the duties of subscribing the bean to the topic or connecting it to the queue based on information placed in the deployment descriptor.

- The *ejb-jar.xml* file contains:
- The fully-qualified class name of the message-driven bean
- A name for the message-driven bean
- The destination type of the bean
- Transaction attributes
- Security information

The following is an example of a typical *ejb-jar.xml* file:

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MyMDB</ejb-name>
      <ejb-class>com.jeffhanson.ejb.MyMDB</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
      <security-identity>
        <run-as-specified-identity>
          <role-name>system</role-name>
        </run-as-specified-identity>
      </security-identity>
    </message-driven>
```

```
    </enterprise-beans>
</ejb-jar>
```

2. Overview of JMS

A message-driven bean (MDB) is an EJB that functions as a JMS message consumer. Unlike session beans or entity beans, clients cannot access message-driven beans directly. Also, unlike session beans and entity beans, a message-driven bean does not have remote or home interfaces. The only access a client has to a message-driven bean is through a JMS destination (topic or queue) of which the message-driven bean is listening.

A MDB must implement two interfaces:

[1] *javax.jms.MessageListener--*
This interface defines the *onMessage* callback method. When a message is put on the queue/topic, the *onMessage* method of the message-driven bean is called by the EJB container and passed the actual message.

[2] *javax.ejb.MessageDrivenBean--*
This is the EJB interface that contains the EJB lifecycle methods:
*ejbCreate()--*called by the EJB container when the message-driven bean is created
*ejbRemove()--*called by the EJB container when the message-driven bean is destroyed or removed from the EJB pool
setMessageDrivenContext(MessageDrivenContext context)-- called prior to *ejbCreate* and passed the message-driven context by the EJB container

The context has runtime information such as transaction data.

The diagram in Figure 1 illustrates the interactions between a JMS message, a client, a topic, an application server, an EJB container, and message-driven bean instances.

As mentioned before, message-driven beans do not have remote or local interfaces as with session beans and entity beans. Message-driven beans are not located by client classes, and client classes do not directly invoke methods on them. All access to a message-driven bean is through a JMS topic or queue which directs messages at the message-driven bean through the EJB container. The EJB container ultimately passes the JMS message to the message-driven bean through the bean's *onMessage* method. All message-driven beans must implement the *javax.ejb.MessageDrivenBean* and *javax.jms.MessageListener* interfaces, as the example illustrates.

Message-Oriented-Middleware provides a common reliable way for programs to create, send, receive and read messages in any distributed Enterprise System. MOM ensures fast, reliable asynchronous electronic communication; guaranteed message delivery, receipt notification and transaction control.

The Java Message Service (JMS) provides a standard Java-based interface to the message services of a MOM of some other provider.

Messaging systems are classified into different models that determine which

client receives a message. The most common messaging models are:

- Publish-Subscribe   Messaging
- Point-To-Point   Messaging
- Request-Reply   Messaging

Not all MOM providers support all these models.
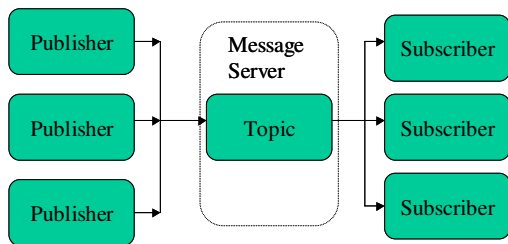
Publish-Subscribe Messaging

Figure 1 Publish subscriber Messaging

When multiple applications need to receive the same messages, Publish-Subscribe Messaging is used. The central concept in a Publish-Subscribe messaging system is the Topic. Multiple Publishers may send messages to a Topic, and all Subscribers to that Topic receive all the messages sent to that Topic. This model, as shown in Figure 1, is extremely useful when a group of applications want to notify each other of a particular occurrence.

In Publish-Subscribe Messaging, there may be multiple Senders and multiple Receivers.

Point-To-Point Messaging

When one process needs to send a message to another process, Point-To-Point Messaging can be used. However,

this may or may not be a one-way relationship. The client to a Messaging system may only send messages, only receive messages, or send and receive messages. At the same time, another client can also send and/or receive messages. In the simplest case, one client is the Sender of the message and the other client is the Receiver of the message.

There are two basic types of Point-to-Point Messaging systems. The first one involves a client that directly sends a message to another client. The second and more common implementation is based on the concept of a Message Queue. Such a system is shown in Figure 2.
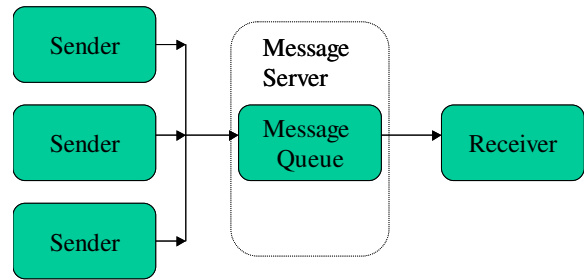

Figure 2. Point to Point Messaging

The point to note in Point-to-Point messaging is that, even though there may be multiple Senders of messages, there is only a single Receiver for the messages.

JMS Parent	Publish Subscribe Domain	Point To Point Domain
Destination	Topic	Queue
Connection Factory	TopicConnection Factory	QueueConnection Factory
Connection	TopicConne	QueueConnection

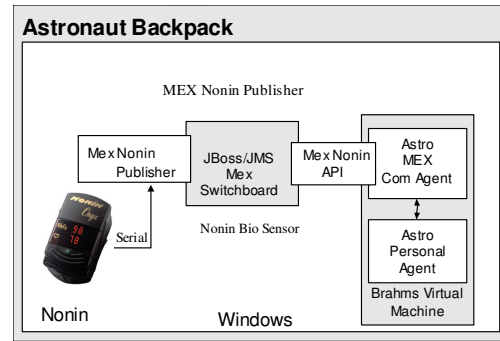|	ction	
Session	TopicSession	QueueSession
MessageProducer	TopicPublisher	QueueSender
MessageConsumer	TopicSubscriber	QueueReceiver, QueueBrowser

Figure 4. Biosensor architecture

3. ARCHITECTURE DESIGN

We selected Publish Subscribe architecture for our data distribution. In our project requirements, the data is distributed to multiple remote clients and the publisher may publish the data to a remote machine. The requirements are satisfied with this architecture.
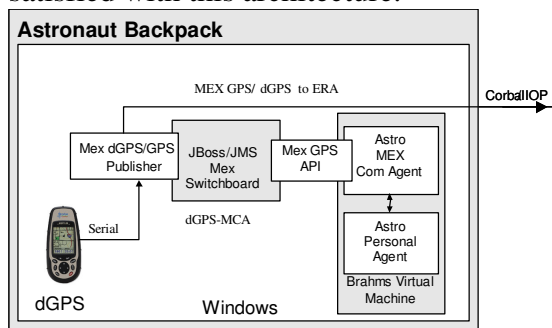
Figure

As Figure 3 shows, the Astronaut carry a backpack and the software is running on the computer in the pack. The GPS unit is connected to the computer and the data is distributed to the JMS server by using the GPS server model. The client will access the data by subscribe to the topic using the API provided by the GPS server developer.
The Bio information of the Astronaut is also distributed by the architecture as shown in the Figure 4.

We are collaborating with a Robotic Rover team (ERA team) for the exploration field test and need to distribute sensor data to their server so the robot will be able to do some action like, following astronaut, take picture at certain point that is related to astronaut or just take a picture of an astronaut. Since our other team (ERA team) is using CORBA framework [6] for their distributed object model, we have to distribute the data across CORBA object by connecting our CORBA client with sensor and push the data to the Rover object running on CORBA ORB.

The architecture of the data distribution to ERA server is shown in Figure 5. The ERA has a server called Executor to accept the data and store in the local memory for the period of time. We need to push the data in the rate that keep the data refresh before the memory time out.
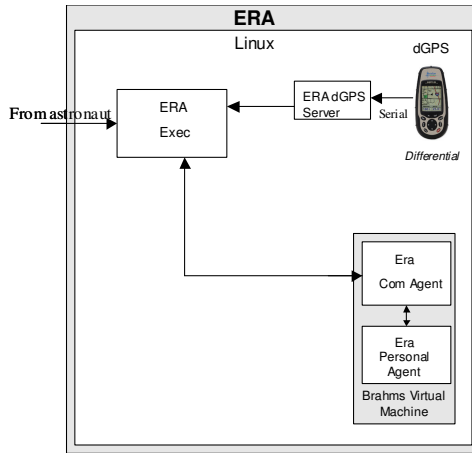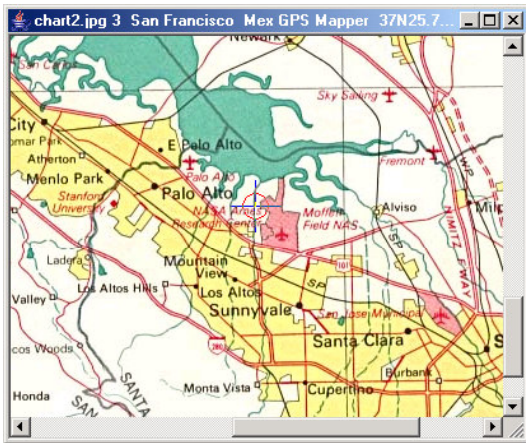
Figure 5. The ERA CORBA server.

The subscribed client will receive the stream of data by intercept the message listener. One of such client is the Rover monitor. It can show the movement of rover on the map in the real time. The figure 6 shows the monitor screen when we did a test in Moffett Field, CA.

The circle with cross is the moving cursor to show the rover location by interpreting the coordinates received from subscripting to GPS topic.

4. THE RELIABILITY AND ROBUSTNESS

The architecture mentioned above has the capability of reliability and robustness. However, some issues are not covered in the architecture itself. The most concerned to us is the network connectivity in the severe environment, lack of the power transmission of the wireless network in the field. We have the simulation test in the Mars Desert Research Station in Utah. The network has outage in the point that the astronauts or the rover is out of the wireless signal coverage. And it will recover when they move back to the place where the signal is strong enough. The short-term network outage is a problem that makes data distribution unreliable. Either the connection loss from the astronaut to JMS server or from the ERA to the JMS server machine, the data distribution will not be able to reach the destination. We took a software measure to overcome the problem; each time when network connection lost, we retry the connection until the connection recovered. The retry is performed in the way that the data is not waiting for the retry to come back, instead, we set a counter that will set to a period of time, like 5 second, as a interval that retry will be attempt. In this way, we do not tight up CPU time, neither do the data resource.

In the subscriber model, we also do the retry to overcome the network outage problem. The Logic is as follows:

```
In the processing of data loop
If (reconnectCounter==0)
        DoReconnect();
        Publish();
Else
        ReconnectCounter--;
Endif


When (ConnectionException)
        SetReconnectCounter;
```

End loop

The other issue is that when use SerialConnection class to acquire data from the COM port, it can not be halt for other tasks since the nature of the data stream come in from COM port is real-time continuers. So we have a separate thread to do the data distribution and this SerialConnection class is dedicated to acquire and store the data in the memory for further processes.

5. CONCLUSION

The field tests and experiments show that the distributed components model that utilized JMS architecture is very suitable for the real time sensor data distribution. It produced the reliable and robust data stream to multiple clients in real time. The publish subscriber model is very scalable even for a large amount of sensors data process. When we have multiples sensor data to be published, the multiple message beans can be created and different topics also can be easily created. The network failure can be easily avoided by writing extra software routine.

REFERENCES

[1] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec,"The Many Faces of Publish/Subscribe", *ACM Computing Surveys*, Volume 35, Issue 2, pp 114-131, June 2003.

[2] L. Garces-Erice and E.W. Biersack and P. Felber and K.W. Ross and G. Urvoy-Keller. "Hierarchical Peer-to-Peer Systems", *Parallel Processing Letters*, Volume 13, Issue 4, December 2003.

[3] Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, and David Swanson, "Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems", *IEEE Transactions On Parallel and Distributed Systems,* Vol. 14, No. 11, November 2003.

[4] Angelo Corsaro, and Douglas C. Schmidt, "The Design and Performance of Real-Time Java Middleware", *IEEE Transactions On Parallel and Distributed Systems,* pp1155-1167, Vol. 14, No. 11, November 2003.

[5] Charles Zhang and Hans-Arno Jacobsen, "Refactoring Middleware with Aspects", *IEEE Transactions On Parallel and Distributed Systems,* pp1058-1073, Vol. 14, No. 11, November 2003.

[6] Victor Fay-Wolfe, Lisa C. DiPippo, Gregory Cooper, Russell Johnston, Peter Kortmann, and Bhavani Thuraisingham, "Real_Time CORBA", *IEEE Transactions On Parallel and Distributed Systems,* Vol. 11, No. 10, October 2000.

[7] Wenbing Zhao, Louise E. Moser, and P. Michael Melliar-Smith, "Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA", IEEE Transactions on Dependable and Secure Computing, pp 14- 23, Vol. 2, No. 1, January-March 2005.