A Distributed Radiation Detection System
Erik Rubin and Chris Knestrick

Abstract
The purpose of this project is to implement a distributed radiation detection system. In this system, multiple sources of data must be connected to an arbitrary configuration of receivers. Both the data sources and the receivers can enter and leave the system at any time, without disrupting other monitoring activities.

The project will be built on two key technologies: Java RMI and CORBA. The goal of this project was to reinforce our understanding of RMI and to introduce ourselves to practical CORBA usage. To facilitate this goal, a small portion of the project was written in C++, and the rest in Java. We used CORBA to bridge the pieces together. CORBA also allowed us to abstract the data sources for the radiation detectors – the C++ code could be gathering its data from any source without the Java side knowing any details.

Summary
Our project is designed to simulate a distributed radiation detection environment. To understand how the project works, we first define the problem that the project is designed to solve.

The environment consists of a disparate group of radiation detectors. Each detector listens to the incoming radiation data for a particular site. There may be many radiation detectors, each tracking a completely different radiation data set.

There is also a set of "radiation clients". Each client represents a user who desires to observe one or more of the radiation detectors. A client does not know a priori what radiation detectors are present on the network.

Thus, the problem is: How can a client become aware of radiation detectors and then request to receive incoming data from a specific detector?

Now that we have described the problem, let us discuss our approach to solving the problem. We addressed this problem with a combination of technology from Java and CORBA. The attached diagram shows the architecture of our project. The central component of the project is the Radiation Server. The server acts as a registry for the radiation detectors. A detector registers with the server when it comes online, and unregisters when it goes offline. The clients can then query the server to determine what detectors are present in the network. A client can request to receive packets of information from any radiation detector. Once a detector becomes aware that a client wishes to receive packets, the detector creates a new port and waits for the client to connect to the port. Then, a direct connection is used to communicate rather than having to use the central server to distribute the packets.
Communication between the server and the detectors and clients is achieved using RMI. Communication between detectors and clients for broadcasting packets occurs through direct socket connections. Finally, CORBA is used in the detectors to provide and interface to a C++ class that actually provides the radiation data. Using CORBA provides the flexibility of receiving the raw data from any source, even a source that directly access hardware via C++ use.

We can now discuss how the project turned out. The most interesting twist that the project took is that CORBA had a very steep learning curve. The key idea behind CORBA is that there is an "Object Request Broker", or ORB, that provides the services necessary for remote object communication. The client interacts with the ORB to discover remote objects, and to activate instances of those objects.

The remote object can be implemented in a variety of languages. A special language, the "Interface Definition Language", or IDL, is used to describe the attributes of the remote object to the client application. An IDL Compiler uses the IDL description of the remote object to create a "stub". The stub looks just like a local reference to the remote object in the native language of the client. The stub communicates with the remote object through the ORB, transparently to the client program.

For the scheme described above to work, there needs to be an ORB installed on all of the machines which will act as object servers or clients. To complicate matters, there are many different ORBs from a variety of vendors, and even some freeware or GRL'ed ORBs. Thus, we were faced with choosing an ORB, installing it, and then using it. Just installing an ORB and working with the sample code took several weeks.

We chose the freely available OmniORB as our ORB for the C++ portion. The ORB that comes as part of the Java Development Kit was used for the Java portion of the project. We installed it on a Windows machine and, after working with some simple examples, we wrote an object in C++ that generates a stream of pseudo-random numbers. Then we compiled the IDL that describes this object so that we could access the object through Java. Finally, we created a Java class that looks to other Java users just like a local pseudo-random number generator, but it really is using the C++ based generator. This was a big step because it showed how we could use CORBA to abstract away exactly where the radiological data comes from. It could just have easily been some C++ code wired to a detector that was generating the numbers.

The RMI portions of the code were built in much less time. As we indicated before we started this project, we had some RMI code to work with as a result of a past project, and this provided us with a good base on which to build. It is fortunate that we had this foundation to start with, because the CORBA integration took a lot longer and was more difficult than originally anticipated.

The RMI part of the code is built just as envisioned in the introduction. There is a central server, called the “RadiationServer” that listens on a port. When radiation
detectors start, they register with the server. Clients who log into the server receive a periodically updated list of detectors who are registered with the server. The client can ask to receive packets from a detector. Once the request is brokered through the server, the client and server make a direct socket connection for packet transmissions. This prevents the server from being overloaded and it means that connections already established would continue even if the server was temporarily down. Thus, a measure of fault tolerance is introduced into our system.

END
Radiation detectors register with the server

Client polls the server to find all detectors. The client then selects one to which to connect.
Server informs the radiation detector that a client wishes to connect to it. The detector will create a port on which to listen for a connection from the client. The server then provides the client with the IP/port to connect to the detector.

The client connects to the detector via a TCP/IP socket. At given intervals, the detector broadcasts it’s reading to any clients with which it may be communicating.
The radiation detector is actually two parts. The java front end provides the RMI and socket interface to the server and clients. However, the back end is actually a C++ object which represents the low-level hardware interface (not unreasonable, since such low-level could who more than likely be legacy code). In reality, the hardware interface is nothing more than a random number generator (since there is no real hardware) that generates readings. The C++ object is registered as a CORBA object, and is polled on a timed interval by the java front end for a “reading”.