The SIMSA (Single Interface for Multiple Services Architecture) began as a student research project. The goal of the project was to create a single threaded, server that handles multiple, stated, concurrent socket connections and also allows services to be started and stopped at runtime. The long term goal of the project as a whole is to be adopted by our university's Computer Science program as a teaching tool for network and architecture classes. Since development began, it became obvious that this project has a lot of applications in subjects like service-oriented problems, distributed computing, message brokering (which is really the whole point of the server).

Before looking at the technical details, it's important to understand the terminology in the SIMSA project.
CLIENT: any program that connects to the server via a socket connection to port 12345 and speaks the SIMSA-client XML commands.
MODULE: any program that connects to the server via socket connection to port 12346 and speaks the SIMSA-modules XML commands. Modules that are written in the same language as the server should be able to be dynamically loaded.
ALIEN MODULE: a module that connects to the server without being explicitly instantiated by the server. In our Java server, any module that is loaded outside the VM that the server is running in is an alien. Alien modules can be running on remote hosts.
SERVER: a program that provides two socket connections for modules and clients to connect on. It handles the registration, message brokering and other functions.

SIMSA is an architecture and it defines a communication interface/protocol rather than an actual program. It defines the language and program requirements that define how clients and modules must behave in order to be used with a single interface that connects to multiple services. The API reference is under development and we have a reference implementation of the server written in Java. There are also reference implementations of modules and clients written in various languages proving that the interaction is completely language independent.

The reference server is written in Java because Java has the unique ability to load byte code dynamically without a lot of pain. Java is also a natural choice for platform portability. The final reason we chose Java is the new java.nio package that provides ByteBuffer based SocketChannel and the Selector object that allows multiple concurrent socket connections on the same thread. This project requires Java 1.5 or greater to run. The server is meant to be run as a daemon and provides another socket connection (port 12344) for an administration console.

Communication with the server uses very simple XML and allows for service lookup, name resolution, connection and disconnection. Modules and clients may define their own tags as well as long as they aren't the same as the server tags. Conceptually it is useful to think of the server as a post office. Incoming messages are contained in an XML "envelope", that envelope is read and removed. The message is then passed to the final recipient who actually reads the message and processes accordingly.

Some possible projects include:
1) Game back ends(Module)/networking layer(Server)
2) Chat servers
3) Distributed computing(modules that bridge multiple servers)
4) MIME encoding could provide file transfer
5) Service Brokering
6) etc...