Java Distributed Data Acquisition and Control

User's Guide

Version 2006.03.14

Glenn Engel

Jerry Liu

Glen Purdy

Agilent Laboratories
3500 Deer Creek Road
Palo Alto, CA 94304

Copyright © 2006 Agilent Technologies.

All rights reserved

This document contains information which is protected by copyright. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Agilent Technologies.

Agilent Technologies is a registered trademark of Agilent Technologies. Names of products mentioned herein are used for identification purposes only and may be trademarks and/or registered trademarks of their respective company.

Notice

The information contained in this document is subject to change without notice. Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties or merchantability and fitness for a particular purpose.

Agilent Technologies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Use, duplication or disclosure by the U.S. government is subject to restrictions as set forth in subparagraph ©(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19c(1,2).

Table of contents

1 Introduction 1

2 Context 2
- 2.1 Background 2
- 2.2 Overview 2

3 Installation 6
- 3.1 Introduction 6
- 3.2 Getting the Files 6
- 3.3 Extracting the Files 6
- 3.4 Required Third-Party Packages 6

4 File Organization 8
- 4.1 Directory Structure 8
- 4.2 Package Namespace 8

5 Building Source Code 9
- 5.1 Introduction 9
- 5.2 J2SE Sample Application 9
- 5.3 J2ME Sample Application 10
- 5.4 J2EE Server Application 10
- 5.5 Additional Notes 11

6 Setting up Eclipse 12
- 6.1 Local Archive 13
- 6.2 CVS 19
- 6.3 Running your first application 26
- 6.4 Setting up your own application 30

7 Database Setup 35
- 7.1 Introduction 35
- 7.2 Hqsldb 35
- 7.3 MySQL 35

8 Getting Started in JDDAC with JddacClient Application 37
- 8.1 Introduction 37
- 8.2 Getting Started 38
- 8.3 Next Steps 42

9 Running the Server 44
- 9.1 Starting the Server 44
- 9.2 Server web interface 44
- 9.3 Probe Configuration 51
- 9.4 Debug Pages 52

10 Object Model 54

11 Data Model 56

12 Measurement Model 57

13 Probe Model 58

14 Communication Model 59
14.1 Publish-Subscribe 59
14.2 Client-Server 59
14.3 Messages 60
14.4 Transports 60

15 Naming Model 61
15.1 Overview 61
15.2 Jddac URI and RFC2396 62
15.3 Implementation 62

16 Thread Model 63

17 Probe-Server Interaction 65

18 Configuring Applications with XML 66
18.1 Probe 66
18.2 Server 69

19 Dynamic Configuration with Server GUI 77

20 Database Storage 78

21 NRSS Data Feed 79

22 CSV Data Feed 80
22.1 Overview 80
22.2 Request Parameters 80
22.3 Data Returned 81
22.4 Incremental Measurement Retrieval 81
22.5 Example 82

23 JDDAC web services Interface 83
23.1 Introduction 83
23.2 Server 84
23.3 Probes 84
23.4 Measurements 85

24 Probe Startup Sequence 66

25 Probe Measurement Flow 88

26 Server Startup 89

27 Server Measurement Flow 91

28 Adding an F-Block 92
28.1 Introduction 92
28.2 A New F-Block 93
28.3 Configuring F-Blocks 94
28.4 Multiple F-Blocks Example 95

29 Adding a Transducer 98
29.1 Introduction 98
29.2 Overview 98
29.3 A New Transducer 98
29.4 Configuring Transducers 101

29.5 Using Transducers 101
30 How to Add Web Pages to Server 103
31 Introduction to Measurement Calculus 89
32 Sample Applications 105
32.1 Traffic Monitoring 105
32.2 NetBEAMS 106
33 XML Schema 107
34 List of Blocks 110
35 Detailed Block Descriptions 110
35.1 ConditionEval 112
35.1.1 Condition Types 113
35.2 Reporter 112
35.3 NCAP 116
35.4 BasicTransducer 116
35.5 BasicTIM 116
35.6 GenericStore 116
35.7 GenericArgArrayStore 116
35.8 MeasTableStore 116
35.9 MetadataStore 116
35.10 ProbeCheckin 116
35.11 ReportDBAccess 116
36 FAQ 117

1 Introduction

This book is about a software application platform called JDDAC (pronounced jay-dak), an acronym which stands for "Java Distributed Data Acquisition and Control". JDDAC is a Java based platform designed to help developers build applications to interact with our physical environment. JDDAC has its origins in industry standards for smart transducers. Over the last few years, JDDAC has been used in application domains ranging from oceanographic research to automobile traffic monitoring. It has also received the Duke's Choice Award at the annual JavaOne conference for technical innovation in the Java space. Best of all, JDDAC is currently publicly available under the BSD open source license, which makes it amenable for inclusion within commerical products.

This book is intended to be used in a number of different ways. First, it provides an introduction to the art of distributed measurements and sensor network design. It discusses the new sensor network systems that have been made possible by hardware advancements in recent years, and the software challenges which face someone who is looking to deploy such systems. In this part of the book, we discuss our philosophy in dealing with these obstacles and how the various tools in the JDDAC toolbox can help someone overcome these problems.

The second part of the book deals with using a JDDAC system. For a someone with a measurement problem she needs to solve (say an oceanographer who wants to know what is happening beneathe the waves), we show how she can use JDDAC to assemble a measurement software application with no programming. In this section, we talk about the logistics of setting up and configuring a JDDAC application.

The third part of the book, and the longest section by far, deals with developing a JDDAC application. In this section, we provide the details needed by someone customizing the basic JDDAC platform for his own particular measurement needs. This section is the canonical programmer's guide with topics such as the basic object, communication, and data models. It also contains short tutorials for specific tasks, such as how to attach a new sensor to the system or how to introduce a new measurement processing functionality into an application.

The fourth part of the book introduces sample applications and case studies. This section provides an overview as to the kinds of systems that JDDAC was built to support. We describe several real-life applications that are built on the JDDAC platform, and take the reader step by step through the choices that were made in the course of system design. In this section, the reader will be able to see how information provided in the previous section can be used weaved together to build a scalable and robust measurement application.

We hope the book will appeal to a varied audience with different backgrounds. For a technical minded person interested in sensor networks and distributed measurements, this book provides an introduction to this burgeoning field and highlight some of the challenges which currently face the practioners. For someone with a need to acquire data from the physical environment, this book shows how JDDAC can be used as a toolkit to solve these problems. And for the software developer interested in creating his own sensor network by building on the foundation offered by JDDAC, this book provides the map which unlocks the secrets of the inner workings of JDDAC.

In this document, the symbol indicates an informational note or constraint.

The symbol indicates a point of caution.

2 Context

2.1 Background

Java Distributed Acquisition and Control (JDDAC) is a collection of Java Application Programming Interfaces (API) which connects Java applications to the physical world by providing principled sensing and actuating capabilities for edge devices.

The physical world is a complex place, and there is wide variation in the myriad of sensing devices used to capture the state of the world. These devices often generate rich data contents and are heterogeneous in how they are characterized, how they are accessed, and how the measurements they generate can be interpreted. The data types from these devices varies greatly as well, ranging from a simple scalar value to a multi-dimensional spectrum of vector values.

In building a large-scale pervasive computing system containing edge sensing devices to interact with the physical world, the heterogeneity of these devices, and that of the physical world, pose significant challenges to the system architect. Without a principled approach to interact with the different types devices and their data, such pervasive systems can never scale to handle all these different devices and the data that they generate.

The existing Java infrastructure provides a feature-rich computing and communication infrastructure, but the infrastructure is weaker when it comes to the connectivity to the real world. The various existing Java interfaces provide connectivity access to devices — the lowest layer in the protocol stack — but provide little guidance at higher abstraction levels such as device characterization and data interpretation.

JDDAC fills in a gap at these higher levels by offering principled methods to describe the edge devices, and methods to describe how they and the data they generate can be used by Java applications. In summary, JDDAC offers abstractions to reduce the complexity in dealing with sensors and actuators at the edge of the network. It provides “last mile” functionality between Java applications and the real world.

2.2 Overview

Many advances are making it easier to build measurement systems. Over the last several years, improvements in various technologies such as chemistry, semiconductors, Micro-Electro-Mechanical Systems (MEMS), and others have made cheaper sensors possible. The cost of making measurements has decreased. This is a trend that is continuing with the demand for more information, with the implication that in the near future, we will be seeing more and different instances and types of sensors.

At the same time, improvements in wireless technologies have resulted in a large selection of capable communication mechanisms. These solutions run the gamut of the spectrum, from long-distance high capital cost solutions such as cellular networks, and short distance solutions such as WLAN and Bluetooth, to extremely short range and low cost solutions such those used in RFID and a number of other wireless sensor protocols. As these solutions proliferate and become more common, the costs are coming down as well.

Meanwhile, computation capabilities continue to make it easier to build large systems. On the hardware front, we continue to benefit through smaller and more complex hardware platforms from the trend described by Moore's Law. As the trend continues, it becomes possible to embed computation capabilities in increasing smaller devices for less cost. On the software front, a number of factors are also making it easier to build large software systems. One important factor is the standardization of specifications, environments, and programming languages. As the industry matures, the products and tools are becoming easier to use. Another significant factor is the popularity of the open source movement and the resultant tremendous selection of easily available and high quality open source software such as Linux and MySQL. Because so much of the basic infrastructure is freely available and of high quality, it has become possible for system builders to move up the value chain and focus on application differentiators rather than the basic plumbing.

In many cases, much of the information computation and communication infrastructure have become commodities. The frontiers have moved on to information acquisition and derivation of knowledge and value from the acquired information.

The purpose of JDDAC is to support the construction of large-scale, distributed data acquisition and control systems which embody a wide variety of sensors and actuators. JDDAC helps the system architect to overcome some of the obstacles that arise in building a large-scale distributed measurement and control system.

One of the challenges today in building such a system comes in interacting with a heterogeneous population of sensors, each with its own unique characteristics. A large-scale measurement system may be long-lived and certainly will likely contain many different kinds of transducers. For the system to scale, both in size and complexity, it is vital that the differences in the devices be abstracted and not find their way into the main program logic of the application.

Another challenge comes from the need to coordinate the actions of the different components in the system to address a specific problem. Today, components in a distributed system coordinate their actions via exchanging command and response messages. For many situations, this strategy works well. However, there are scenarios where sending or receiving such messages is costly, such as in the cases of wireless sensor networks. Here, an alternative solution may be desirable.

Once data have been acquired by the sensors in a measurement system, the data values are transformed into measurements which are then used in various calculations and processing operations. The vast majority computing systems simply deal with values such as integers or floating point numbers. But in a measurement system, the metadata associated with the value is essential to making sense of the value and thus is just as important as the value.

In many systems today, the metadata associated with the measurements are implicit in the program logic, and the rules for operating on the measurements are ingrained in the system. This approach works well for measurement systems with dedicated deployment, but does not work well for long-lived dynamic systems. These systems may need to deal with

many different types of sensors which may not be specified at the time the system was built.

Lastly, the types of processing performed in a measurement system are data-centric. Instead of a von Neumann style control-driven computation model where data is represented as a piece of memory being acted upon by sequential instructions, it is often more helpful to model the system in a dataflow computation model, where data traverses the nodes of a graph, with each node representing a data processor.

This computation model has several advantages as applied in our domain. Computation and communication entities are kept relatively distinct and separate, giving the programs a simplicity not found in von Neumann architectures. The execution of the processing nodes are based not on instruction sequence, but on the availability of data, thus making the program execution potentially parallel and asynchronous. This last point characteristics is important because the physical world is, after all, asynchronous as well. Lastly, since there is almost no dependency among the processing nodes, except for the data which flows among them, reusability is high in such systems.

JDDAC addresses the needs within the Java ecosystem for interfacing applications to the physical world. The Java environment, with its large variety of libraries and APIs, focuses primarily on computation and communications, two of the most fundamental aspects in a distributed computing system. Distributed measurement systems add a third dimension: measurement. The Java universe is scarce in APIs for dealing with measurement process, and physical world in general.

A typical distributed measurement system contains the layers as shown in Figure 1. JDDAC addresses the needs of some of the fundamental layers within the system.

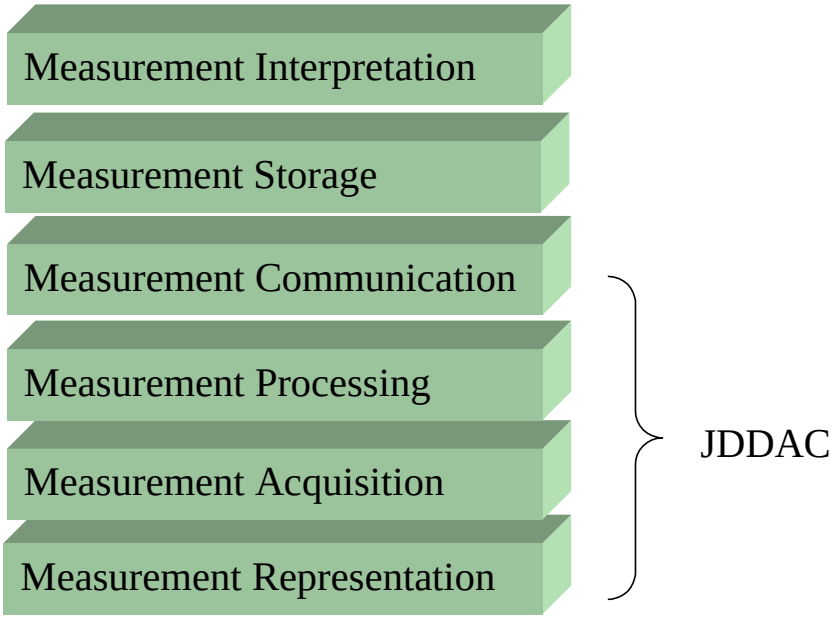


Figure 2-1 Scope of JDDAC in the context of the Measurement Stack

JDDAC is partly based upon guidelines and concepts found in the IEEE 1451.1 [1] and IEEE 1451.2 [2] specifications. IEEE 1451.1 defines a component framework for smart sensors and actuators systems, including an object model, a communication model, and metadata for measurements, among other things. IEEE 1451.2 defines a common interface for transducers to facilitate plug-and-play functionality down at the transducer level.

JDDAC provides several tools to help the Java application developer in building large-scale measurement systems. Starting at the physical world interface, JDDAC defines abstractions for interacting with transducers for data acquisition and transducer actuation based on the IEEE 1451.2 specifications. For addressing the challenges of synchronizing the actions of distributed nodes in a measurement system, it specifies an interface for managing and synchronizing extremely accurate clocks based on the IEEE 1588 specifications [3].

For using sensor measurements, JDDAC provides a flexible measurement data model for accurately representing the physical world and defines set of measurement calculus operations for operating on these measurements. Lastly, drawing from the IEEE 1451.1 specification, it specifies a dataflow oriented computation and communication framework to process measurements.

As an area of future expansion, JDDAC will also be drawing upon the specifications and concepts from the upcoming IEEE 1451.0 specification.

3 Installation

3.1 Introduction

This chapter describes how to install the JDDAC distribution.

The distribution comes in two forms: a binary package and a source package. The binary package contains precompiled Java code for a measurement probe. It is suitable if you simply want to test drive JDDAC. You do not need to have a Java development environment set up. Simply having a Java Runtime Environment is sufficient to run the probe.

The source package contains the actual source code, libraries, and the scripts to compile the JDDAC system. It is suitable if you would like to experiment with making modifications or additions to JDDAC or see how it is built.

The current JDDAC distribution runs under all three versions of Java: J2EE, J2SE and J2ME.

3.2 Getting the Files

The most recent release can always be found at the JDDAC Home Page at http://jddac.dev.java.net. You can either download it as a ZIP archive, or retrieve the files from the CVS server on java.net. See java.net documentation for how to configure your CVS client if you wish to go this route. Unless you think you may want to check files into the CVS repository, we recommend you download the ZIP file at first.

3.3 Extracting the Files

The distribution is in Zip format. On Windows, use Winzip to extract the files from the distribution. On Unix, use the 'unzip' command to extract the files. You can place these file anywhere in your directory tree. In the rest of the document, we will refer to the directory where the files have been extracted as JDDAC_HOME.

3.4 Required Third-Party Packages

JDDAC is based completely on freely available packages and libraries. Most of the libraries needed by JDDAC are included in the distribution. However, there are a few packages which must be installed separately due to licenses which restrict redistribution. These packages include:

- Java Software Development Kit, version 1.5.0 or higher – This is needed to compile and run JDDAC. It is needed for both the binary and source packages.
- Jakarta Ant, version 1.6.1 or higher – This is needed to build the JDDAC distribution. This is needed only if you download the source package and wish to compile the code yourself.

- Java Wireless Toolkit, version 2.2 or higher - This contains the libraries needed to compile J2ME midlets as well as the emulator to run midlets on a host platform.

The Java SDK is downloadable from http://java.sun.com/j2se/1.5.0/download.html. Look for the “Download J2SE SDK” Link. Be sure and follow the installation instructions found here.

Jakarta Ant is downloadable from http://ant.apache.org/bindownload.cgi. You can read the installation instructions from http://ant.apache.org/manual/install.html. Make sure to set the appropriate environmental variables for your specific platform as specified towards the end of the document.

Java Wireless Toolkit is downloadable from http://java.sun.com/products/j2mewtoolkit.

IMPORTANT: After you install the Java Wireless Toolkit, you need to set the environmental variable WTK_HOME to point to the root directory of where the Java Wireless Toolkit is installed.

4 File Organization

This chapter describes how the files in the JDDAC distribution are organized.

4.1 Directory Structure

These are the top level directories:

- apps – Contains JDDAC applications built on the JDDAC APIs. Within the apps directory, there is a subdirectory for each standalone JDDAC application.
- common – Contains core components used by all JDDAC APIs.
- contrib - Contains vendor specific files.
- docs – Contains documentation and javadocs.
- lib – Contains the compiled JDDAC libraries.
- lib-ext – Contains the third party library packages used by JDDAC. Within the lib-ext directory, there is a subdirectory for each third-party package.
- portal – Contains the source code specific for the server (J2EE) platform.
- probejme – Contains the source code specific for the embedded (J2ME) platform.
- probejse – Contains the source code specific for the desktop (J2SE) platform.
- tools – Contains miscellaneous tools used in building JDDAC.

Within the apps directory, there is a subdirectory for each standalone JDDAC application.

Within the source code directories (common, portal, probejme, probejse), there is a *src* directory which contains the source code, and a build.xml ant build file for building the source code in the src directory. The output of the compilation is typically stored in a *bin* directory which is dynamically created.

Within the lib-ext directory, there is a subdirectory for each third-party package.

4.2 Package Namespace

All Java source files are in the namespace net.java.jddac.*.

There are some classes which are used by all JDDAC APIs. These include classes that represent the common data structures and classes that provide basic utilities such as logging. These classes are found in net.java.jddac.common.*

The classes which make up the Java Measurement Calculus Interface (JMCI) are in net.java.jddac.jmci.*.

The classes which make up the Java Measurement Dataflow Interface (JMDI) are in net.java.jddac.jmdi.*.

The classes which make up the Java Transducer Interface (JTI) are in net.java.jddac.jti.*.

All three of these packges have subpackages below them. Please see the javadocs descriptions for additional details.

5 Building Source Code

5.1 Introduction

This chapter provides instructions on how to build the JDDAC distribution and the example application.

The assumption is that your computer satisfies the system requirements described in the installation instructions and that you have already downloaded and unpacked the source code distribution package. Assume, for these examples, that the JDDAC source code has been installed in the directory JDDAC_HOME.

To build all the JDDAC source code into the library JAR files, go to the JDDAC_HOME directory via the command line:

```
cd JDDAC_HOME
```

Be sure and substitute the directory name of where JDDAC is installed in place of the JDDAC_HOME above!

And simply invoke ant

```
ant
```

This will cause all the libraries to be built.

5.2 J2SE Sample Application

Then, to build the application for the J2SE platform, go to the following directory

```
JDDAC_HOME/apps/JddacClient
```

On Linux platforms, the command would be

```
cd JDDAC_HOME/apps/JddacClient
```

on Windows, it would be

```
cd JDDAC_HOME\apps\JddacClient
```

Be sure and substitute the directory name of where JDDAC is installed in place of the JDDAC_HOME above!

Then, simply invoke ant

```
ant
```

This will cause the JDDAC libraries to be compiled, along with the Java classes associated with the sample application. If the compilation completes successfully, you will find a JddcClient.jar in the JDDAC_HOME/apps/JddacClient/lib directory.

You can then refer to the running the sample application guide to learn how to run the sample application.

5.3 J2ME Sample Application

To build for the J2ME platforms, go to

```
JDDAC_HOME/apps/PhoneClient
```

and invoke ant there to cause a PhoneClient.jar and PhoneClient.jad to be generated in the JDDAC_HOME/apps/PhoneClient/lib directory.

5.4 J2EE Server Application

JDDAC server currently comes with support for two databases. hsqldb, which is an embedded database written in Java, and mysql, which is a popular standalone open source database. the libraries for hsqldb are included within the JDDAC package. MySQL needs to be installed separately. If you do use MySQL, you will need to install a version 4.0 or later.

The sample JDDAC server is contained within two directories under /app: /jddacServer and /jddacWar. The jddacServer directory contains the the server configuration files and the execution scripts for the server. The jddacwar directory contains the static and dynamic web pages, as well as any Java source code specific to the server.

If you choose to use hsqldb, you need not do anything more. We recommend that the first time through, you simply run the server with hsqldb.

To use MySQL, you will need to install the MySQL server, create a database, and add a user account for the JDDAC server with read/write access for the database. We recommend naming both the user account and the database 'jddac'.

Then edit apps/jddacServer/etc/startup/dbCon.mysql.xml, and find two instances of:

```
  <arg
n="DbConMgrBlock.dbUrl">jdbc:mysql://localhost/jddac</arg
>
  <arg n="DbConMgrBlock.dbUser">jddac</arg>
  <arg n="DbConMgrBlock.dbPassword">password</arg
```

These lines assume that the MySQL is running on the same machine as the JDDAC server, that the database and the user account are both named jddac, and the password is called 'password'. Change these settings to ones that suit your environment.

Note that there are two instances of these three lines. Both instances need to be edited.

To build the server, invoke ant at /apps/jddacServer.

5.5 Additional Notes

- There is also an Ant build script, build.xml, present at the top level of the JDDAC directory. If you want to build all the JDDAC libraries without building the applications, you can invoke `ant` at the top of the directory tree.

You can invoke `ant clean` to clean out all the compiled binaries in the directory tree. This is true for the overall build script as well as the ones in the apps directory

6 Setting up Eclipse

This chapter describes how to set up the Eclipse IDE to work with JDDAC source code.

The prerequisites are that you have copies of the following installed on your system:

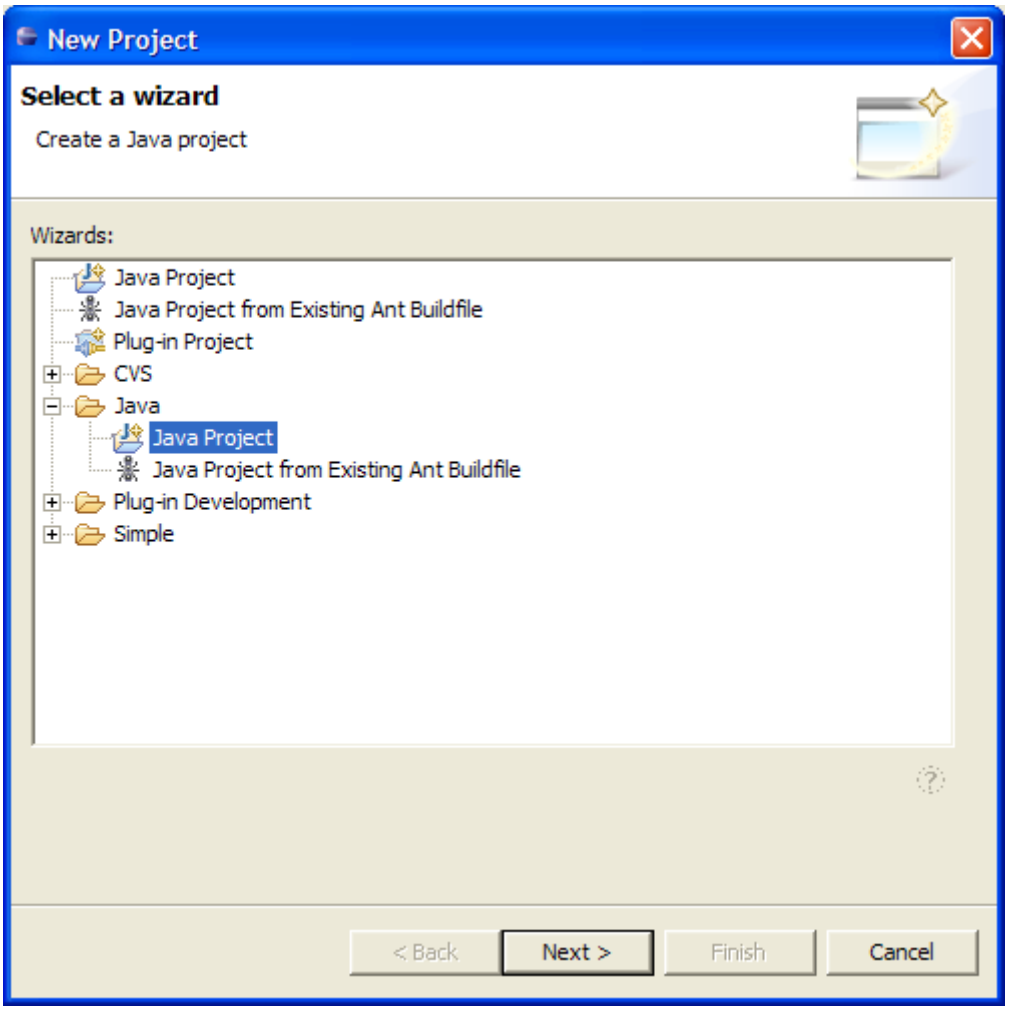
- Java Development Kit 5.0 or later
- Eclipse Version 3.1 or later.
- Java Wireless Toolkit, Version 2.2 or later.
- JDDAC Source Code.

There are two ways to get the JDDAC source code. You can either create a local archive from unpacking the ZIP archive distribution on your system, or you can get it via the CVS server on java.net. Both methods are described below.

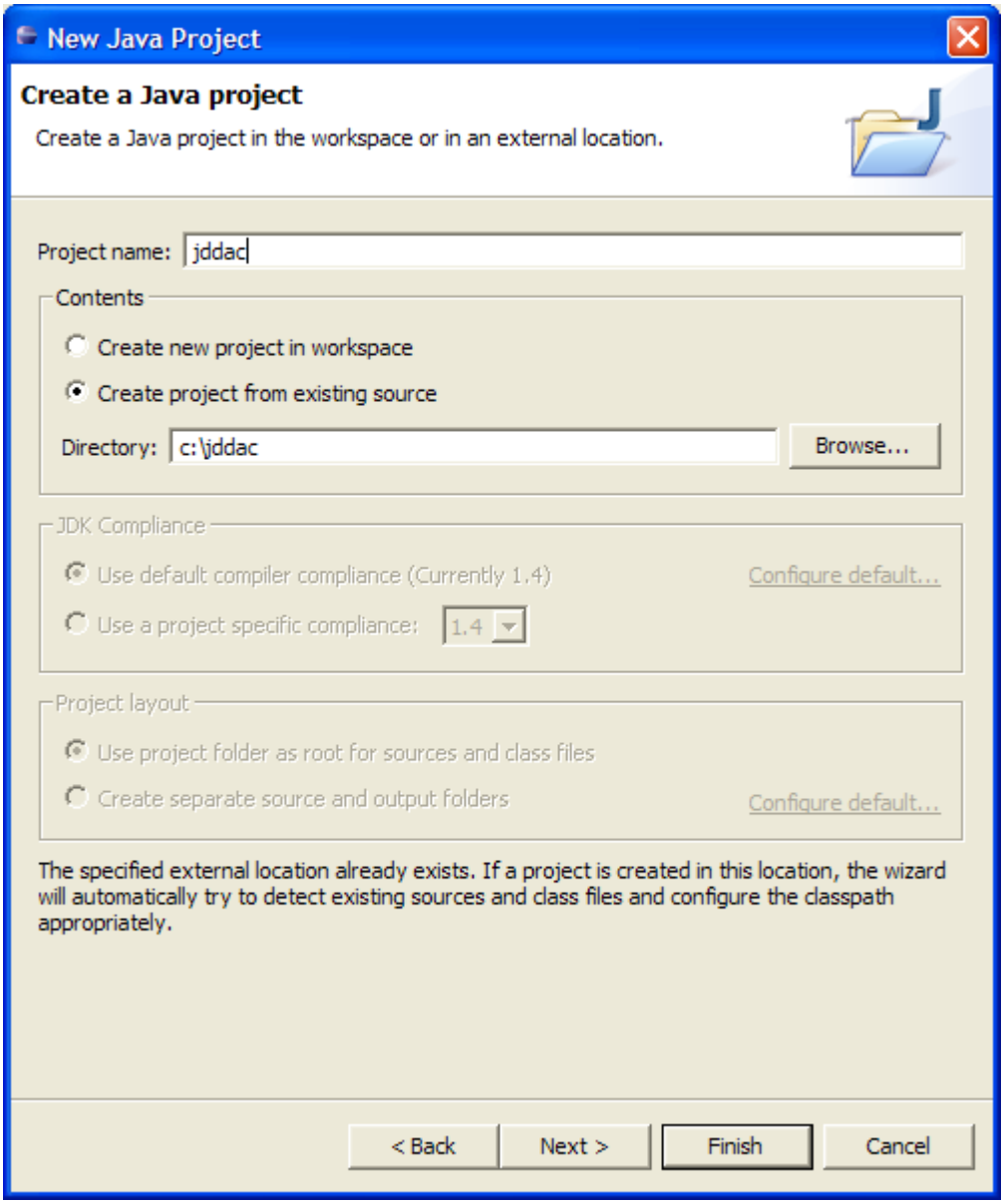
6.1 Local Archive

Assume that you have unpacked the archive at the directory c:\jddac. Wherever you see the path c:\jddac in the examples below, substitute your own path.

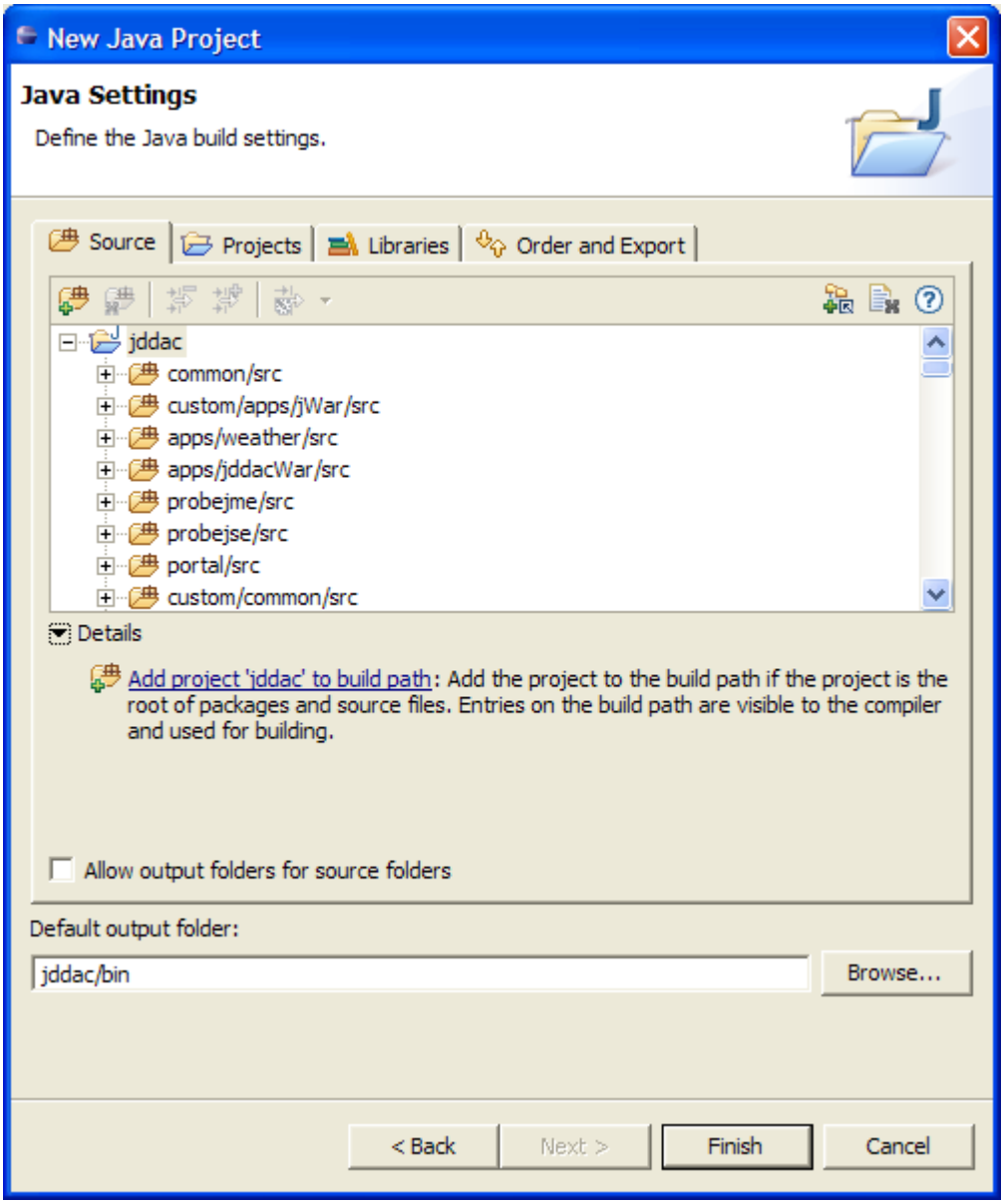
Start Eclipse. Select New->Projects, and select a Java Application project.



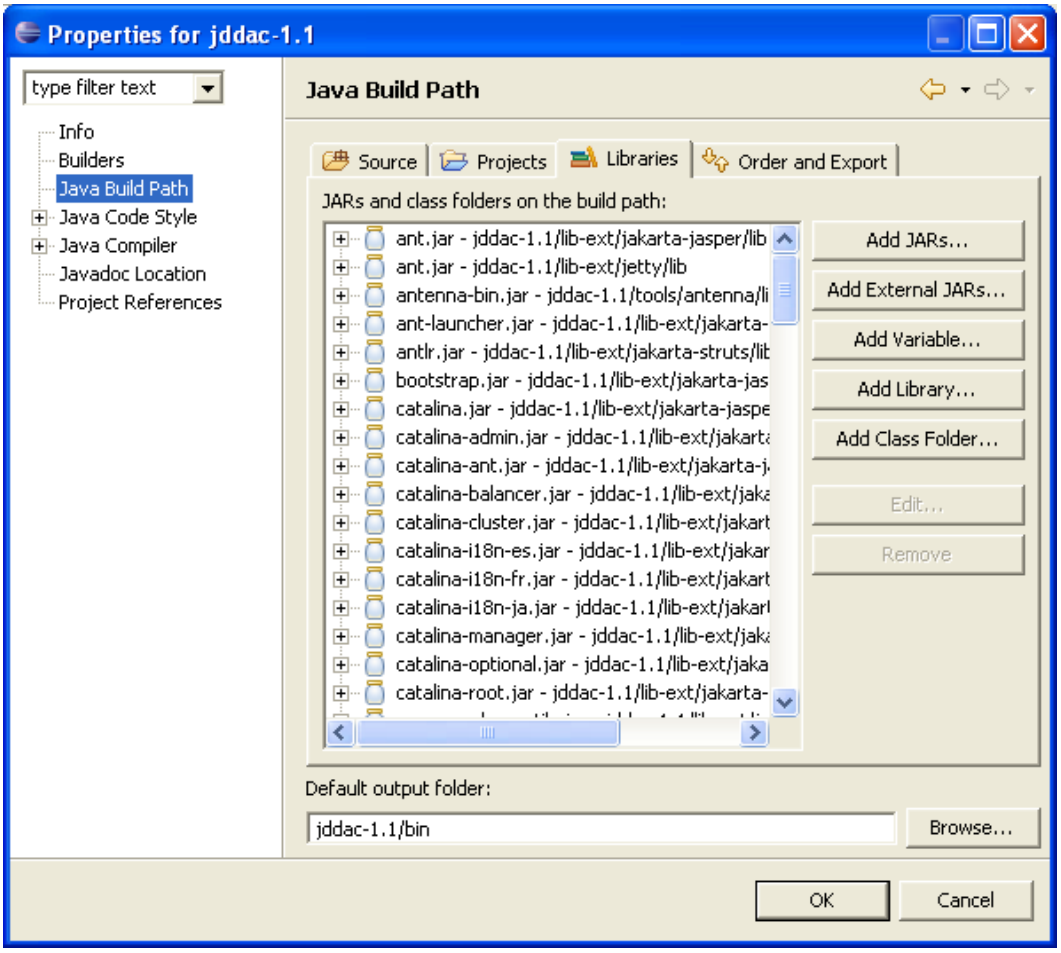
Enter the name of the project and the path to the root of the directory. Use your own path in place of c:\jddac.



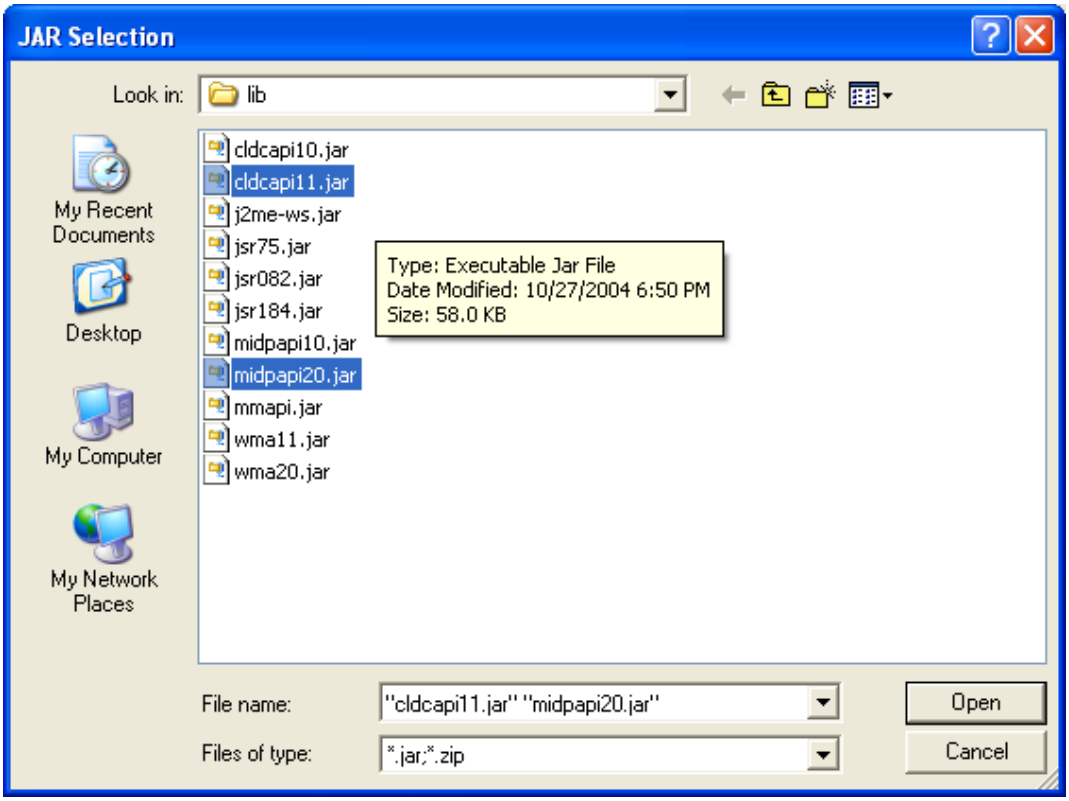
Then click Finish on the next screen. You may have more of fewer directories than what is shown in the screenshot below, depending on the state of the JDDAC project snapshot.



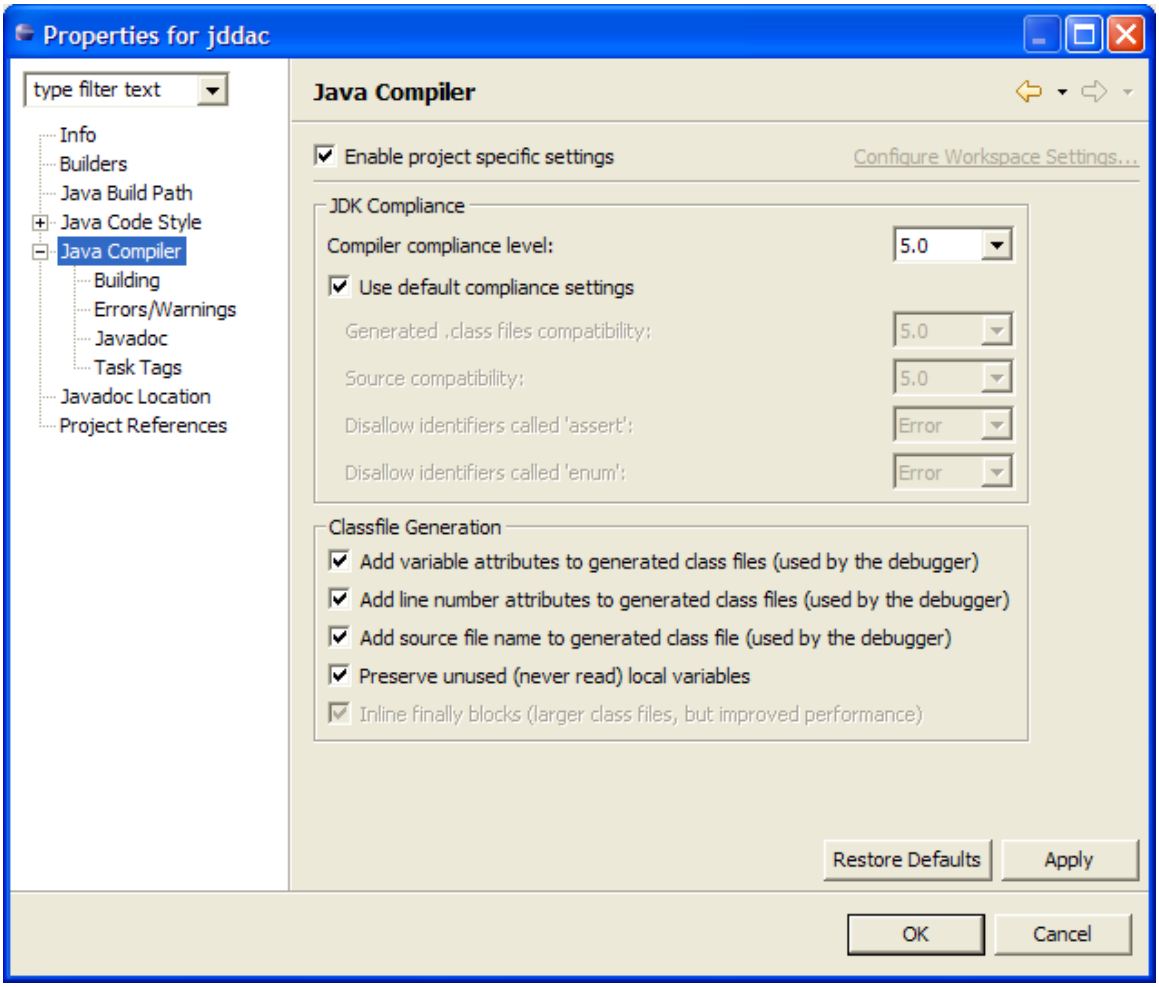
Next, right click on the JDDAC entry on the left, and right click on "Properties" to set project properties. Navigate to Java Build Path, select Libraries, and you should see the following:



Click on "Add External Jars", navigate to where you have installed the Java Wireless Toolkit, and double click on the "lib" directory. You should see a screen like the following. Highlight the two jar files as shown in the diagram and click "Open".



Next, click on the "Java Compiler" option and make sure that 5.0 complicance level is selected. If 5.0 is not available as a choice, you will need to install the J2SE 5.0 SDK on your system before continuing.

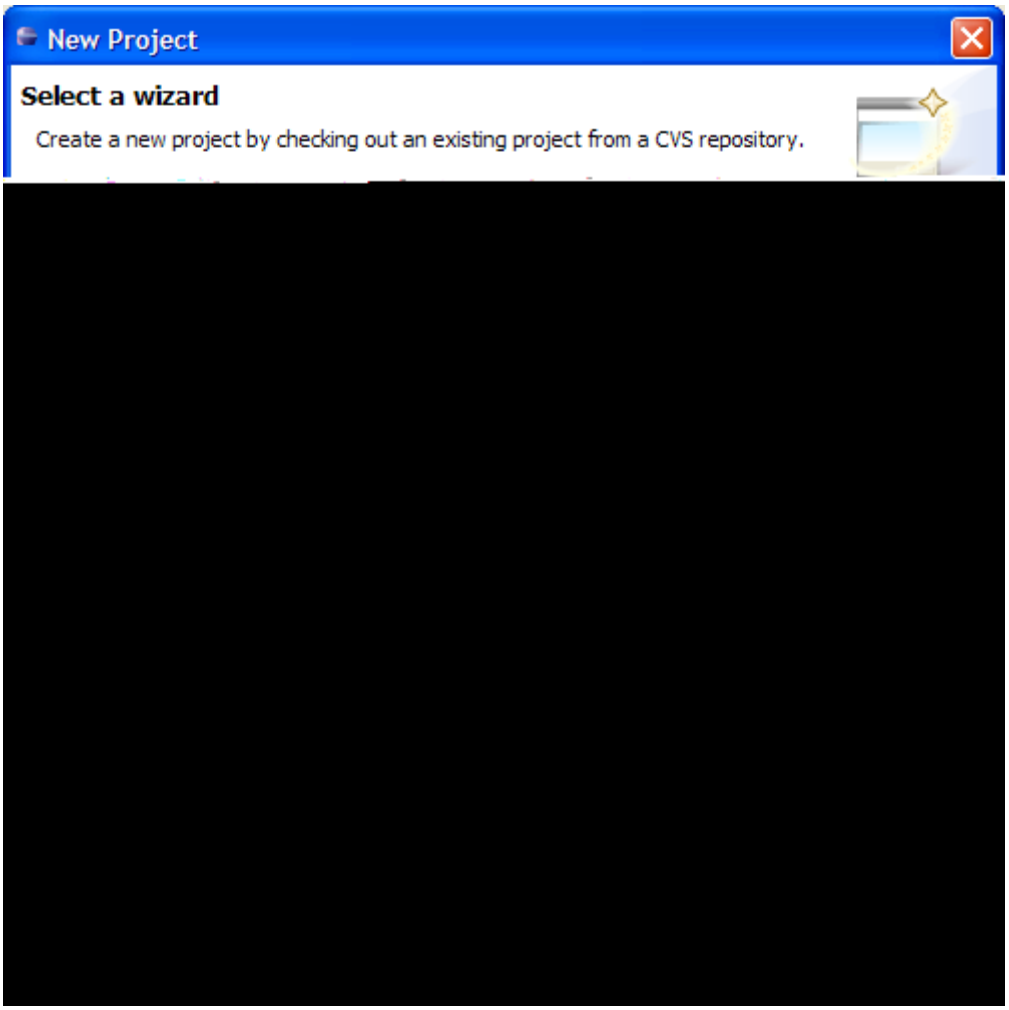


Eclipse will be busy for a while while it rebuilds the workspace (watch the bottom right corner for progress bar). When it's finished, you're all ready to go!

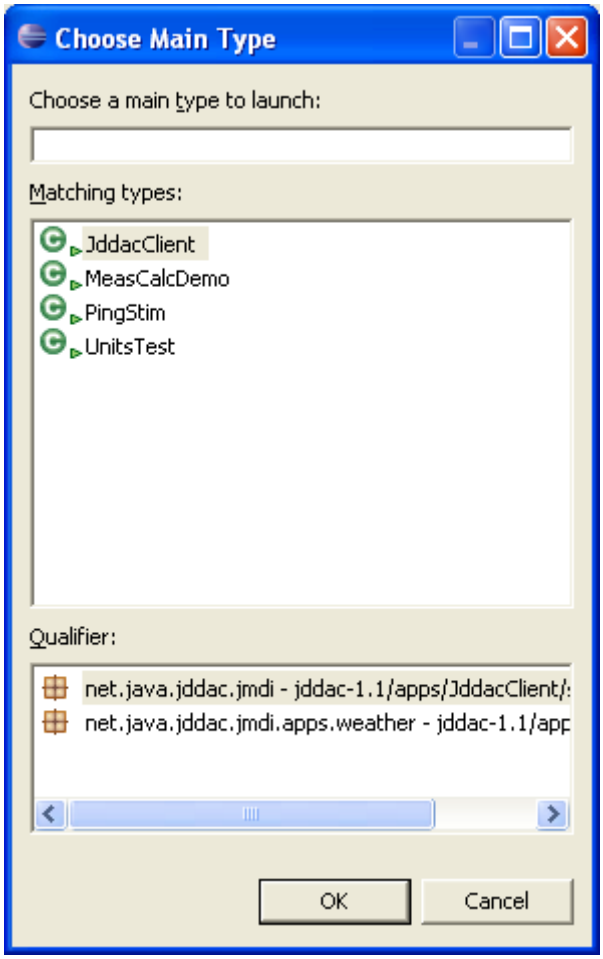
6.2 CVS

This section describes how to create a new Eclipse project drawing source code from a CVS repository.

Select New->projects from the menu on the upper left.

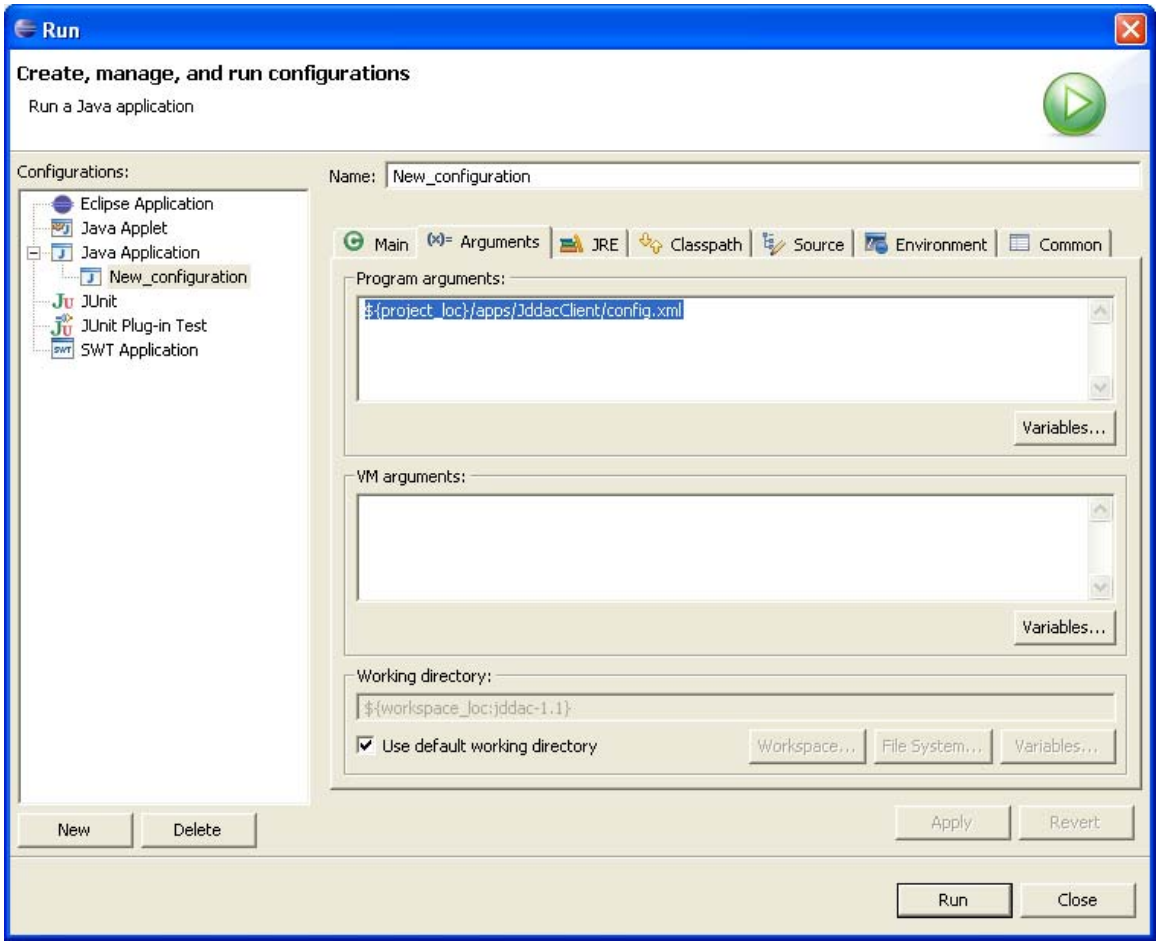


Click on the Search button to view a list of Java applications. Select JDDACClient and click OK.



Next, click on the arguments tab, and enter the following string:

${project_loc}/apps/JddacClient/config.xml

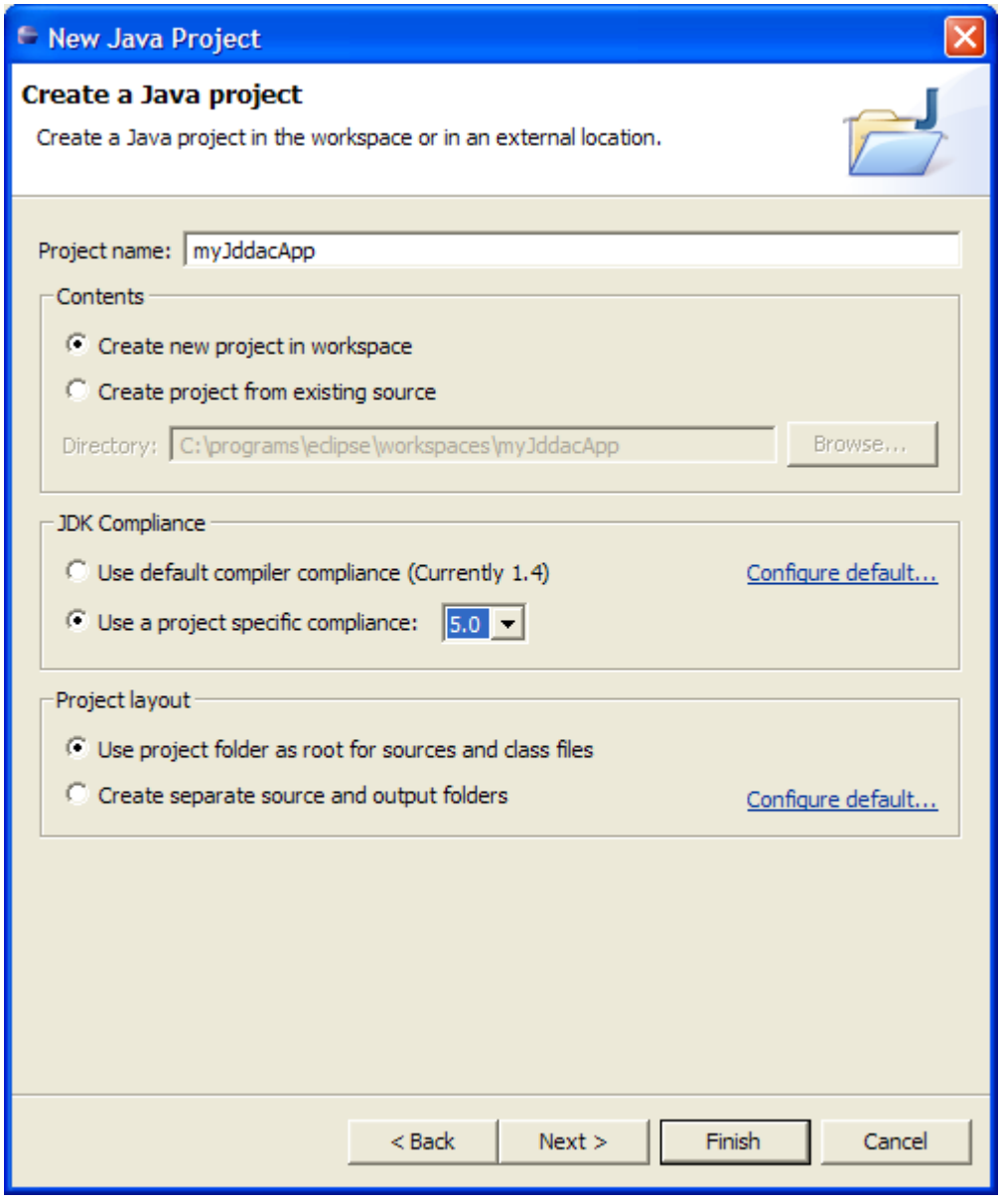


Go back to the Main tab. Click on JddacClient, then click Run. You'll be running your first JDDAC program!

6.4 Setting up your own application

the time will come when you will want to build on the JDDAC core to write your own JDDAC application. Basically, you will create your own Java project, but add Eclipse as a dependency so you will have access to the JDDAC classes and libraries.

First, go to New->Projects, and create a Java application as we did in section 1.2 .Let's call it MyJddacApp.

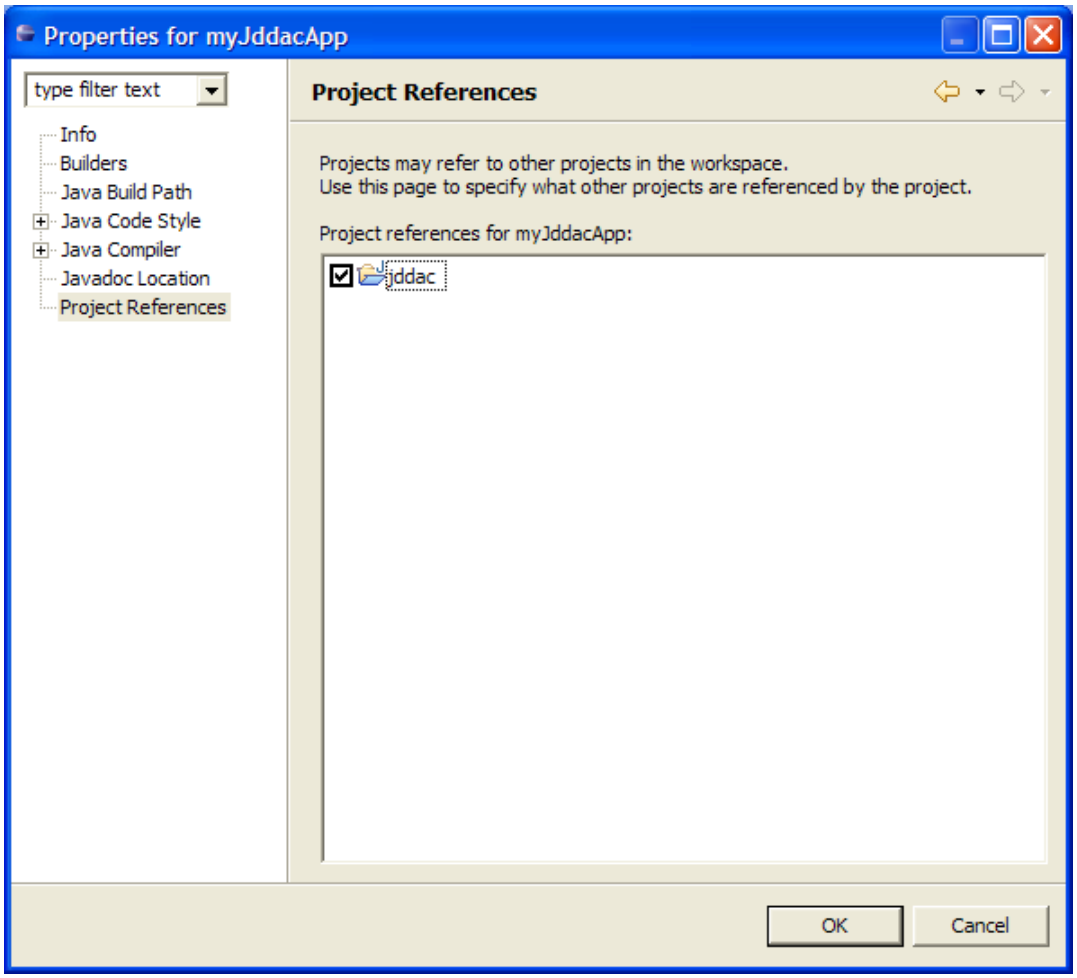


be sure and set the compliance to 5.0 if you are offered a chance to. If not, don't worry. You can set it later.

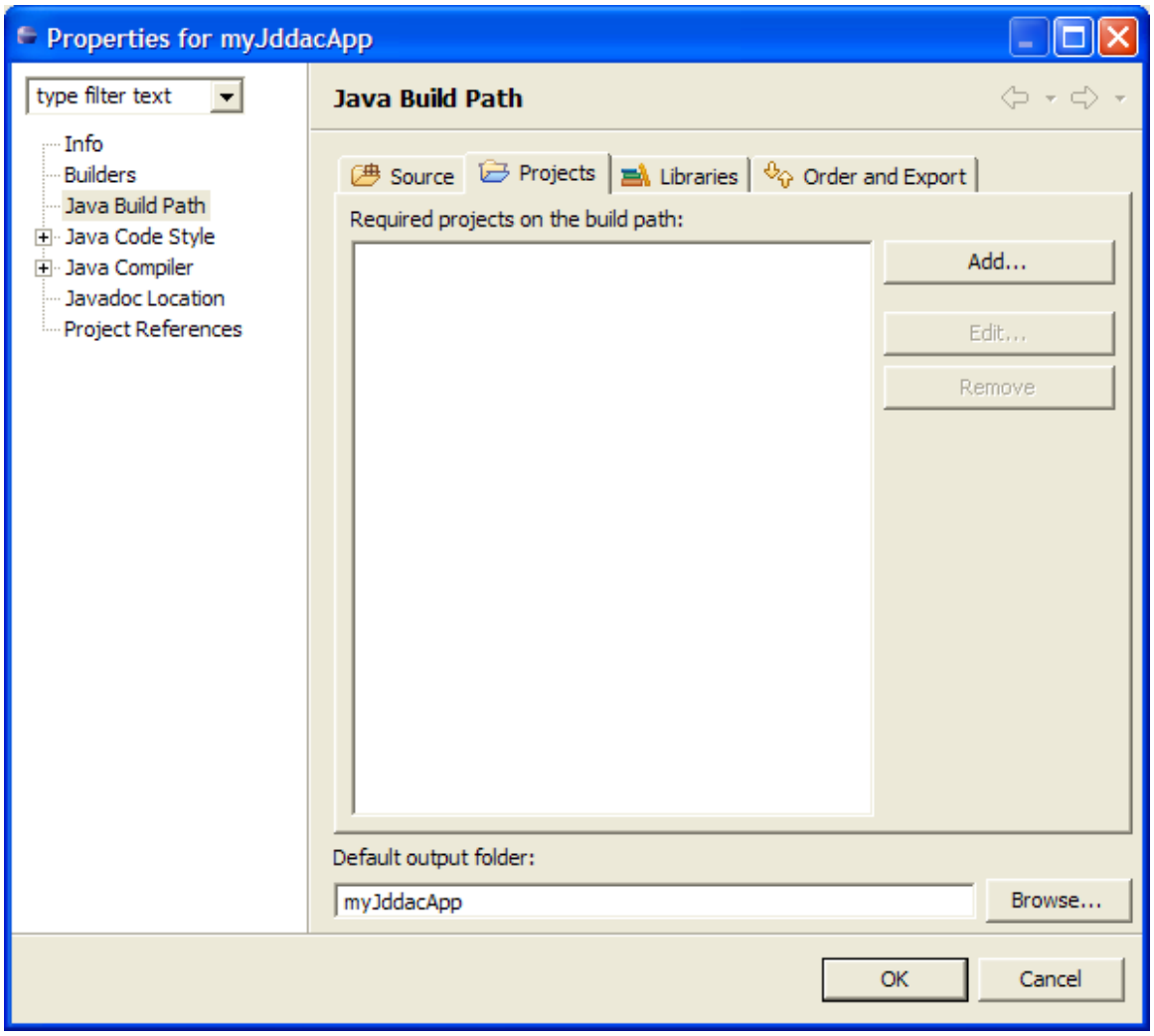
Click through the default settings as in section 1.2 to create your project.

Once the project is ready, we need to add JDDAC as a dependency.

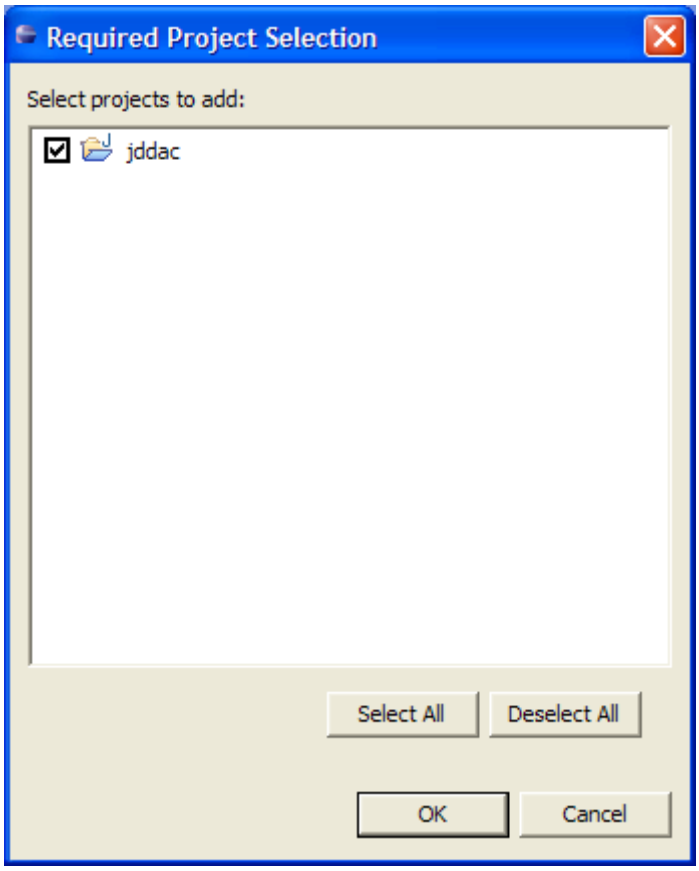
Right click on your project and select "properties'. Click on Project preferences on the left and select jddac.



Next, go to Java Build Path, and click on Add.

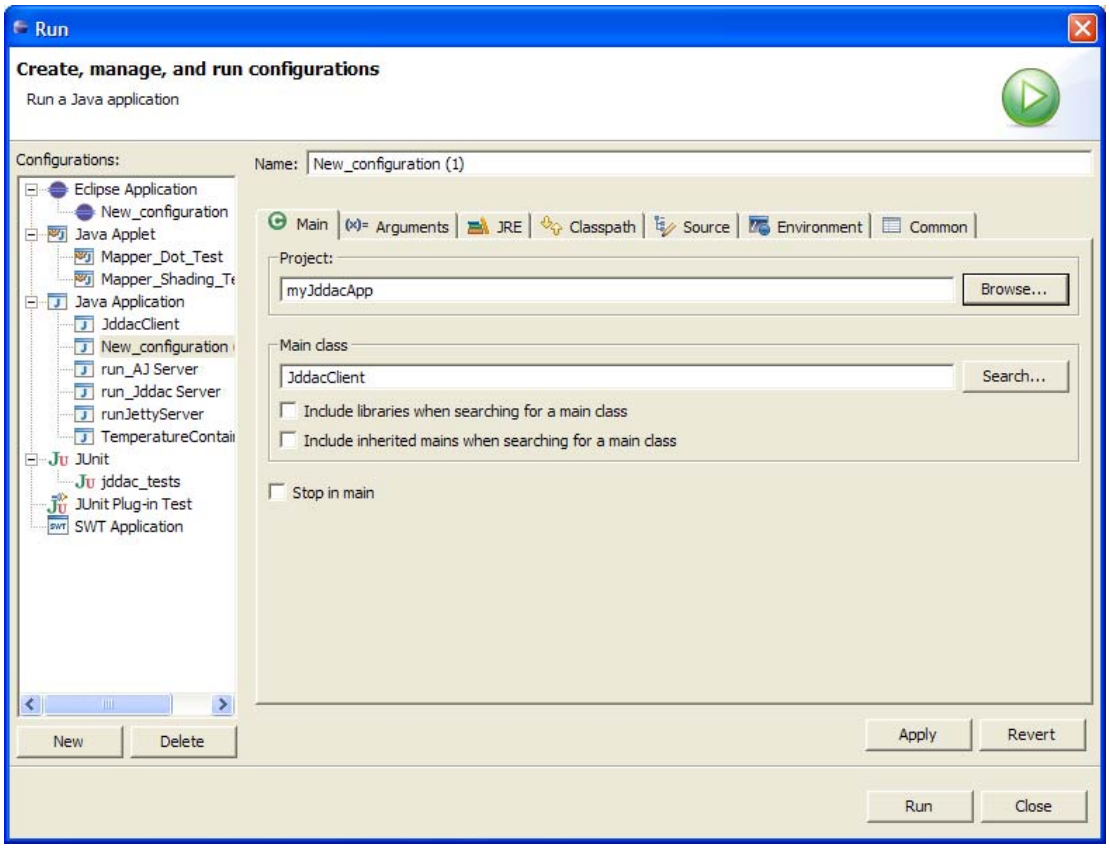


You will be given a chance to select jddac. Do so and click OK.



You are ready now to add code to your project.

When it comes time to run your application, select Run->Run... as we did before. Click on New on the bottom left. Click on Browse to select your project, and Search to select the class as the entry point for your project.



7 Database Setup

7.1 Introduction

This chapter discusses how to set up the database required by the JDDAC server. The information in this chapter is only applicable if you wish to set up your own server.

The JDDAC server uses a relational database to store all the system state information, such as the measurements, system configuration, sensor status, etc. In short, any information that needs to be persisted across power cycling of the server is stored in the relational database.

the JDDAC distribution comes with the open source HsqlDb database preconfigured to work with the JDDAC server. HsqlDb stores the database information in the forms of files within the app/JddacServer directory. HSqlDb is a handy database and can handle small to moderate sized systems. Since it runs in the same virtual machine as the JDDAC server, there are no separate processes that need to be started. And debugging from IDEs such as Eclipse is very simple.

The JDDAC distribution also comes with drivers and configuration files to work with the popular MySql database. MySQL is a very capable database and is suitable for moderate to large systems. the JDDAC public servers are using MySQL as the backend server. The drawback of using the MySQL server is that MySQL needs to be separately installed, configured, and run. But once it has been configured and working, it is capable of handling larger loads than HsqlDb.

JDDAC has been used with other database servers such as Oracle 8i. In theory, JDDAC should work with any database that has JDBC drivers available. Look in the configuration files for the MySQL server to see how to configure JDDAC for other servers.

7.2 Hqsldb

To use Hsqldb as the backend, you do not need to do anything special. Upon detecting an empty database, the JDDAC server automatically populates the Hsqldb with the appropriate schemas. Just execute the JDDAC server from the run.bat or run.sh script from apps/jddacServer/bin and you're all set.

If you ever wish to clean out the contents of the Hsqldb database, simply delete the temp directory in apps/jddacServer.

7.3 MySQL

To use MySQL, you'll need to get a copy of MySQL running on a machine. The details of the procedure is beyond the scope of this document. You'll need to create an account with the privileges to read and write to the database. Add the user account information to the configuration file at apps/jddacServer/etc/startup/dbCon.mysql.xml.

```
<!-- Database Connection Manager for user information -->
 <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.DbConMgrBlock</arg>
  <arg n="instanceName">UserDb</arg>
  <arg n="Entity.description">User DB Manager Block</arg>
  <!-- MySQL DB -->
  <arg n="DbConMgrBlock.dbUrl">jdbc:mysql://localhost/jddac</arg>
  <arg n="DbConMgrBlock.dbUser">jddac</arg>
  <arg n="DbConMgrBlock.dbPassword">password</arg>
  <arg n="DbConMgrBlock.dbDriver">org.gjt.mm.mysql.Driver</arg>
 </argArray>

 <!-- Database Connection Manager for Measurement Data -->
 <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.DbConMgrBlock</arg>
  <arg n="instanceName">DataDb</arg>
  <arg n="Entity.description">Data DB Manager Block</arg>
  <!-- MySQL DB -->
  <arg n="DbConMgrBlock.dbUrl">jdbc:mysql://localhost/jddac</arg>
  <arg n="DbConMgrBlock.dbUser">jddac</arg>
  <arg n="DbConMgrBlock.dbPassword">password</arg>
  <arg n="DbConMgrBlock.dbDriver">org.gjt.mm.mysql.Driver</arg>
 </argArray>
```

Note there are two places where change needs to take place.

Next, edit apps/jddacServer/etc/startup.xml, and change:

```
<!-- Setup database connection -->
 <argArray>
    <arg n="includeFile">../etc/startup/dbCon.hsqldb.xml</arg>
 </argArray>
```

to

```
<!-- Setup database connection -->
 <argArray>
    <arg n="includeFile">../etc/startup/dbCon.mysql.xml</arg>
 </argArray>
```

Lastly, execute these two SQL scripts on MySQL to create the appropriate tables.

- apps/jddacServer/etc/schemaSetup.mysql.sql
- apps/jddacServer/etc/userSetup..mysql.sql

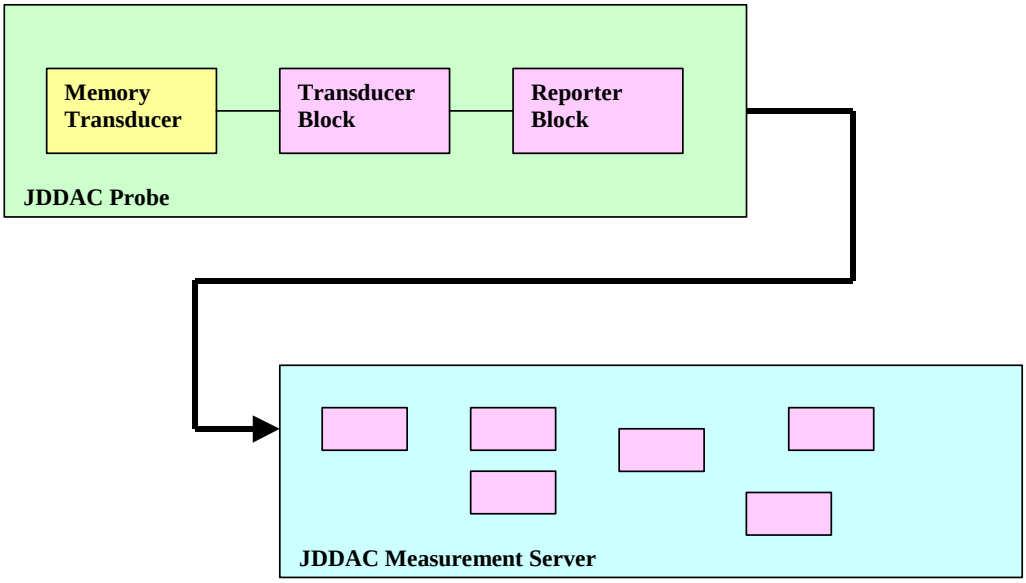
8 Getting Started in JDDAC with JddacClient Application

8.1 Introduction

This note will help you get started with the sample JDDAC application. The sample JDDAC application included in this first release will introduce you to elements of the JDDAC APIs.

This example contains a simple Java measurement probe that runs on J2SE, the most common Java platform. This probe is a measurement dataflow application built on JMDI and JTI. The measurement probe makes measurements and then sends the measurements to a data archive where they are stored.

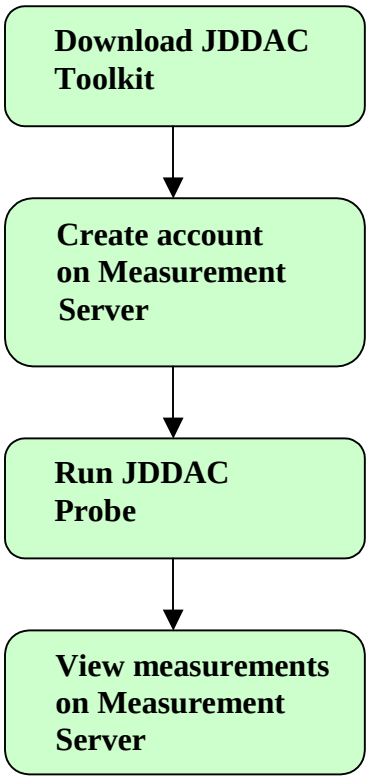
In this example, the dataflow application is very simple. The dataflow network is composed of two blocks: a Reporter Block and a Transducer Block. The Transducer Block is connected to a virtual transducer. The virtual transducer we have included in this distribution makes measurements about the amount of memory available in the Java virtual machine.The transducer is virtual in the sense that it is not connected to any physical measurement devices, but like a physical transducer, it produces measurement data. This way, you won't need to have any sensors connected to your computer to run this example.



This release only contains the JDDAC probe source code because only the J2SE version of JDDAC is included in this release. The source code for the J2EE versions used to build the Measurement Server will be available in the next release. However, there is an operational server available for use with this example. For the purposes of this example, you can focus on the workings of the green box above.

8.2 Getting Started

These are the steps in this tutorial. First, you will install the JDDAC package. Next, you will go to the Measurement Server and create an account. When the account is created, you will be allocated data storage space on the server and issued an authentication key for the probes. Then you will run the JDDAC probe and let it make the measurements and report data. Lastly, you can view the data at the Measurement Server.



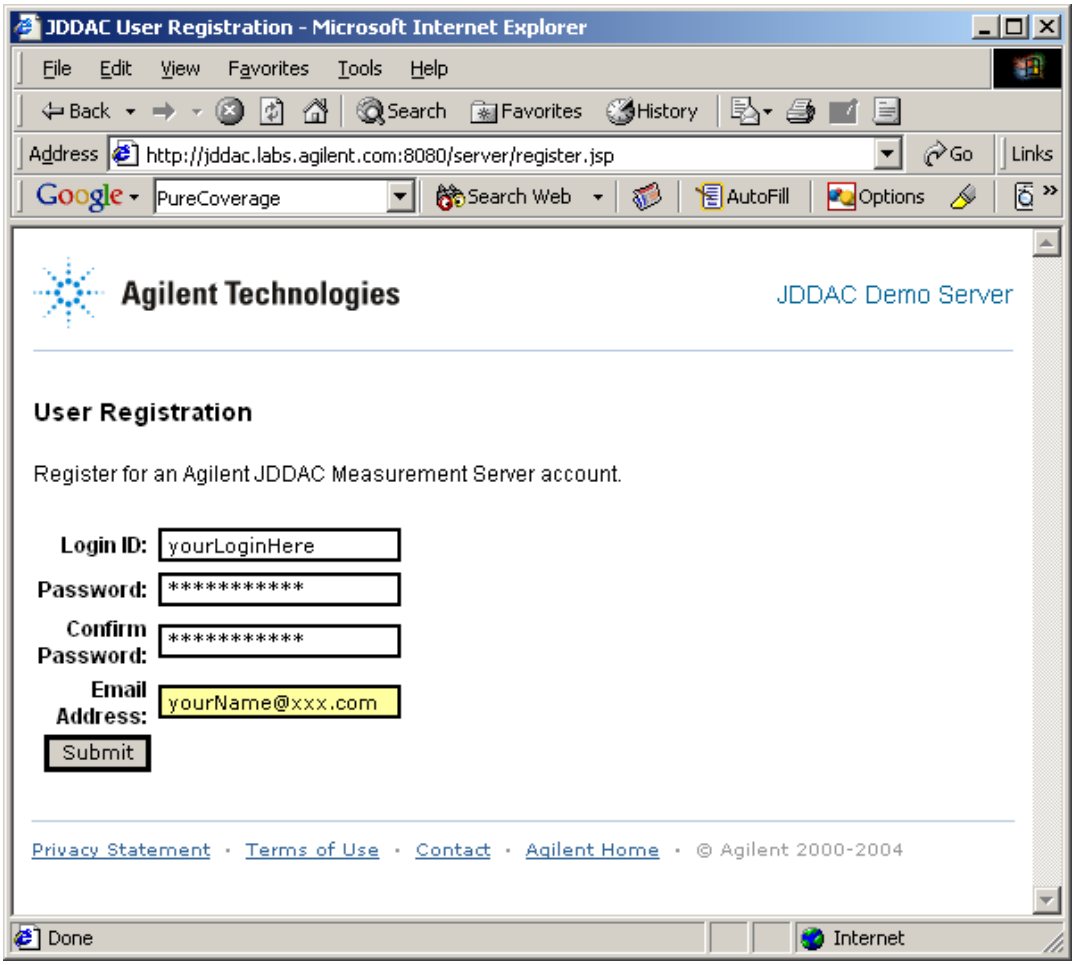
First, make sure that you have working JDDAC installation. If you just want to play with the simple JDDAC probe for now, get the binary package. That way, your installation is ready to go out of the box. You can always get the source package later when you are ready to build your own measurement system.

Read the installation guide to get your system installed.

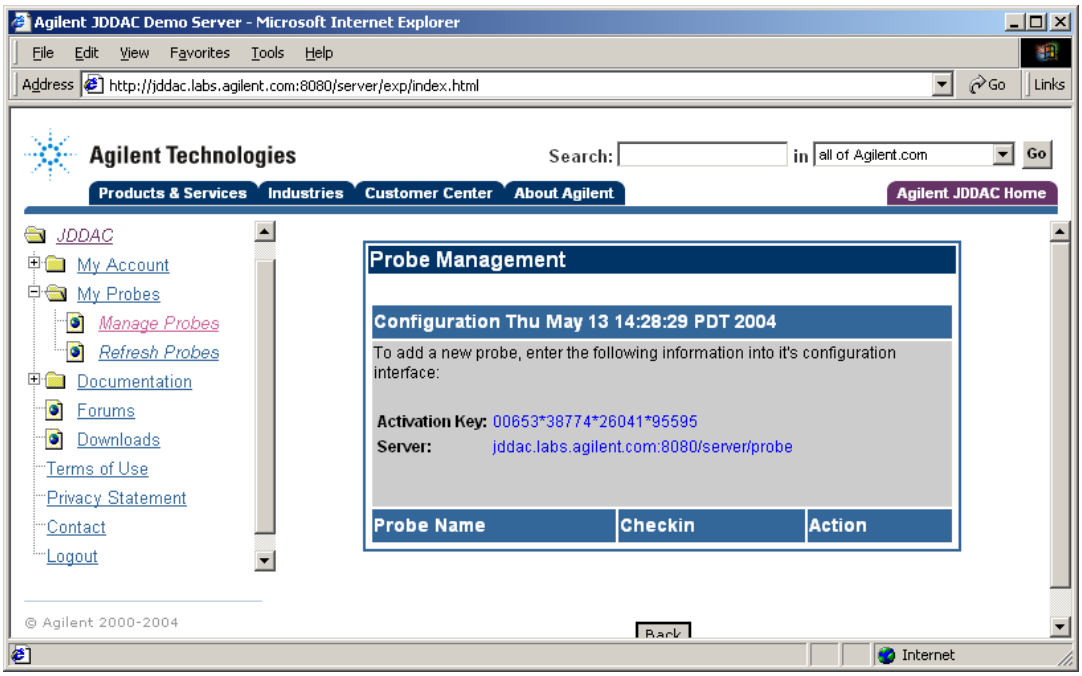
Next, go to the Measurement Server at

http://jddac.labs.agilent.com

and register for an account. No personal information is collected. Only an e-mail address is requested and this is so that we can get you your password should you forget it.



Once you sign in, click on the 'Manage Probe' link on the left to get the authentication key for your account. You will need to enter this information into your probe. The Measurement Server uses the key to authenticate probe measurement data and make sure other people can't view your data.



Now that you have the authentication key, go back to your local JDDAC installation. Let's refer to the root of your JDDAC installation as JDDAC_HOME. Go to the directory: JDDAC_HOME/apps/JddacClient, and start the JddacClient by the command:

java –jar lib/JddacClient.jar

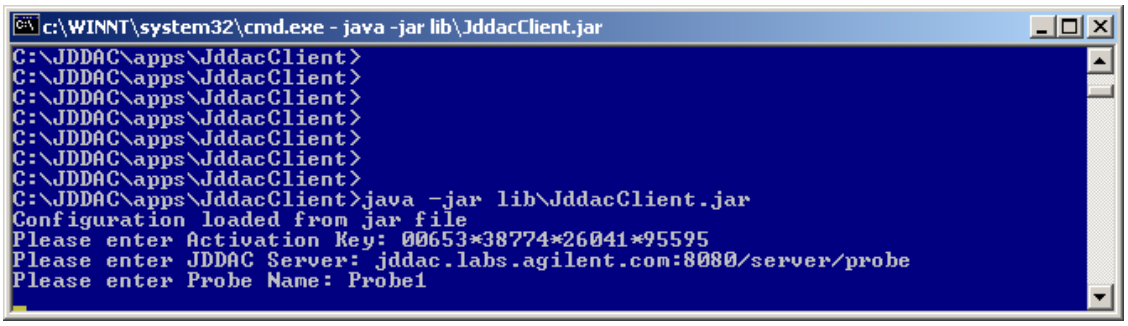
If your network requires use of a web proxy server, you will need to add additional arguments to the command to specify your proxy settings:

java –Dhttp.proxyHost=yourproxy.example.com –Dhttp.proxyPort=8080 –jar lib/JddacClient.jar

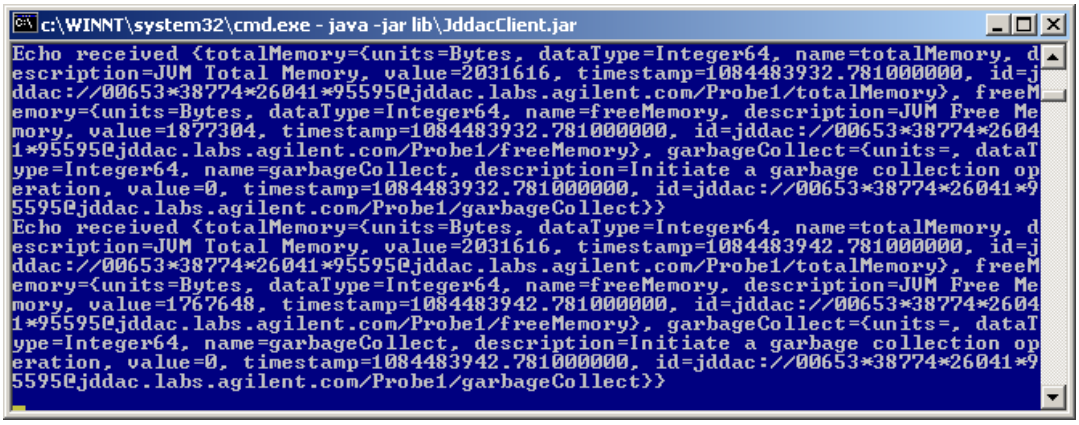
If your proxy requires user authentication, you'll need yet more arguments:

java –Dhttp.proxyHost=yourproxy.example.com –Dhttp.proxyPort=8080 –Dhttp.proxyUser=username –Dhttp.proxyPassword=password –jar lib/JddacClient.jar

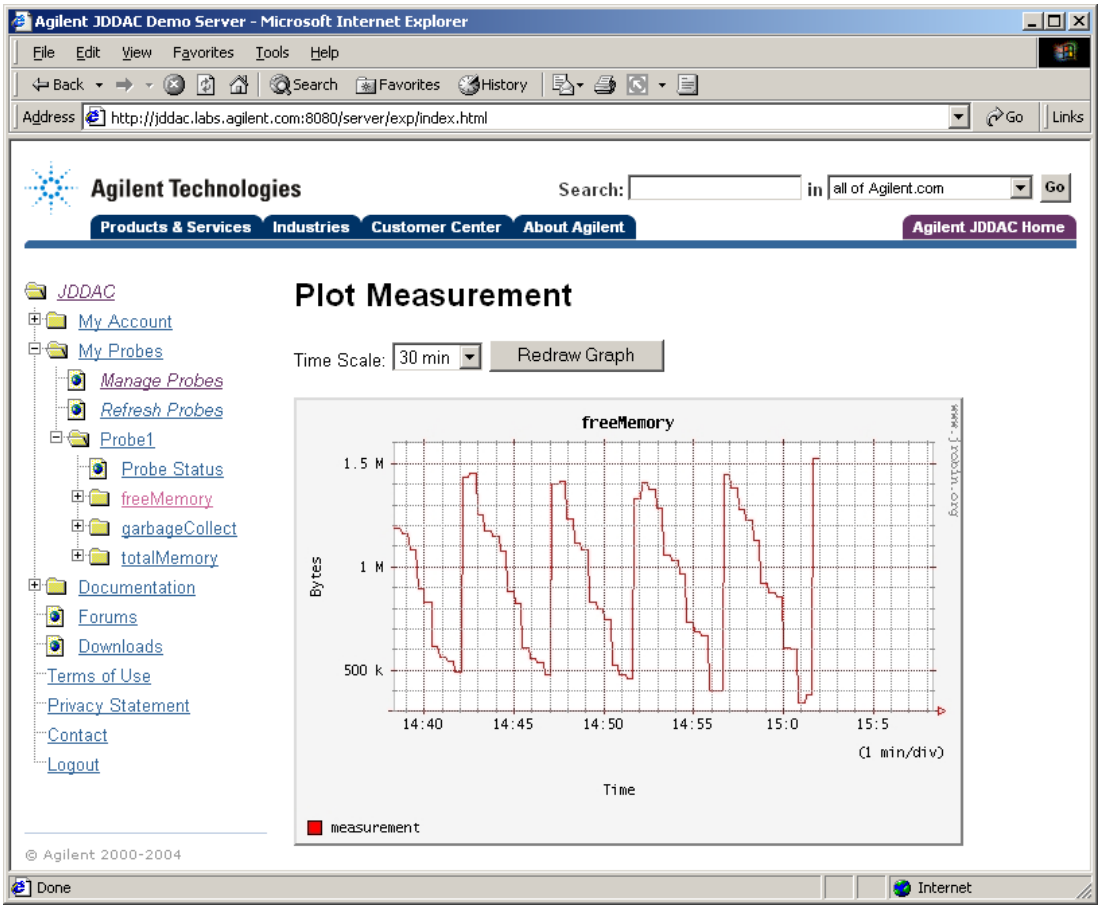
The JDDAC Client application will be started using the configuration from the "startup.xml" file. You will be prompted for the authentication key (enter the one you were issued), the server (jddac.labs.agilent.com:8080/server/probe), and a name (any name you choose to pick). Once you have entered these parameters, the probe will automatically generate a "jddac.xml" startup file for subsequent invocations. The screen may look something like this:



By default, the probe makes a memory measurement every 10 seconds, and the graph on the server updates only once a minute, so wait for a bit and go browse the JDDAC forums on java.net while you wait for some measurements to be made. As measurements are made, they are printed locally:



Next, on your browser, go back to the JDDAC measurement server at http://jddac.labs.agilent.com, log in if your session has timed out, and click on “My Probes” to see your probe. You may need to click “Refresh Probes” if your new probe doesn’t show up in the list. You can now click around and examine the data that your probe is collecting. Use the tree in the left navigation pane to traverse your data sets, and click on the name of the data sets to see a graph of the stored data.



In the example application, the probe generates a measurement record with two sensor measurements *freeMemory*, *totalMemory*. The *freeMemory* measurement measures how much free memory is available in the Java virtual machine running the JDDAC probe, and the *totalMemory* measurement measures the total memory available in the virtual machine.

If you'd like, you can start a few different probe instances with the same authentication key. You will see different data sets appear in the navigation tree when the new probes start reporting their data. In this case, you can specify a configuration file and a probe name on the command line used to start the probe, such as:

java –jar lib/JddacClient.jar jddac.xml Probe2

This way, you can start many instances of the same probe type (JddacClient.jar) yet configure them differently.

8.3 Next Steps

Now that you have the basic example probe running, you can try reconfiguring the probe. JDDAC applications are configured via XML files. Use your favorite editor and open the jddac.xml file in the JddacClient directory. Look for the line towards the bottom of the file that contains:

```
<arg t="int" n="BasicTransducerBlock.period">10000</arg>
```

This is the sampling period of the transducer block. It is set to 10 seconds (10,000 milliseconds). You can change this to some other number and watch the data updated faster on the server.

To do this, you'll need to stop your probe (control-C will do the trick), change the jddac.xml file, and restart the probe again.

You can peruse the Configuration Guide to learn how the configuration works and what the rest of this stuff in the file is.

Unfortunately, reconfiguring this simple example for different sampling period probably isn't very exciting. You'll probably want to make your own custom measurements or wire in your own blocks in this three-block network to perform your own processing.

For example, you may have a One-Wire® device connected to your PC that you may want to hook into this measurement system. Or perhaps you have some processing you'd like to do with the measurement such as checking for threshold on the measurement and firing an alarm if necessary. These actions will require writing new Transducer objects (for the former case) or new Function Blocks (for the latter case). See the tutorials on adding new Transducer objects and one on adding new Function Blocks to the system.

As always, feedback is very welcome and greatly appreciated. Our goal is to make it easy to build measurement systems that span from the sensors to the enterprise. Please post any comments or questions to the JDDAC discussion forum on java.net. Thanks!

9 Running the Server

9.1 Starting the Server

To run the server, you will need the following packages:

- Java Development Kit 1.5 or later, including Java Runetime Environment 1.5 or later
- Java Wireless Toolkit 2.2 or later

To start the server, execute either run.bat (if you're on Windows) or run.sh (if you're on Linux) from the directory /apps/jddacServer/bin

By default, the server runs on port 8080 which is the standard default for java servlet containers. Should you wish to run the server on another port, edit /apps/jettyServer/etc/jetty.xml, find the line

```
        <Set name="Port"><SystemProperty name="jetty.port"
default="8080"/></Set>
```

and change the 8080 to whatever port you desire.

Once you start the server, you will see a number of diagnostic messages scroll by. If all goes smoothly, you should see at the very end a line similar to:

INFO: Started org.mortbay.jetty.Server@19b49e6

(the string of numbers at the end will vary from run to run...)

This indicates that the server is now up and operational.

9.2 Server web interface

You can access the server through its web interface via browser.

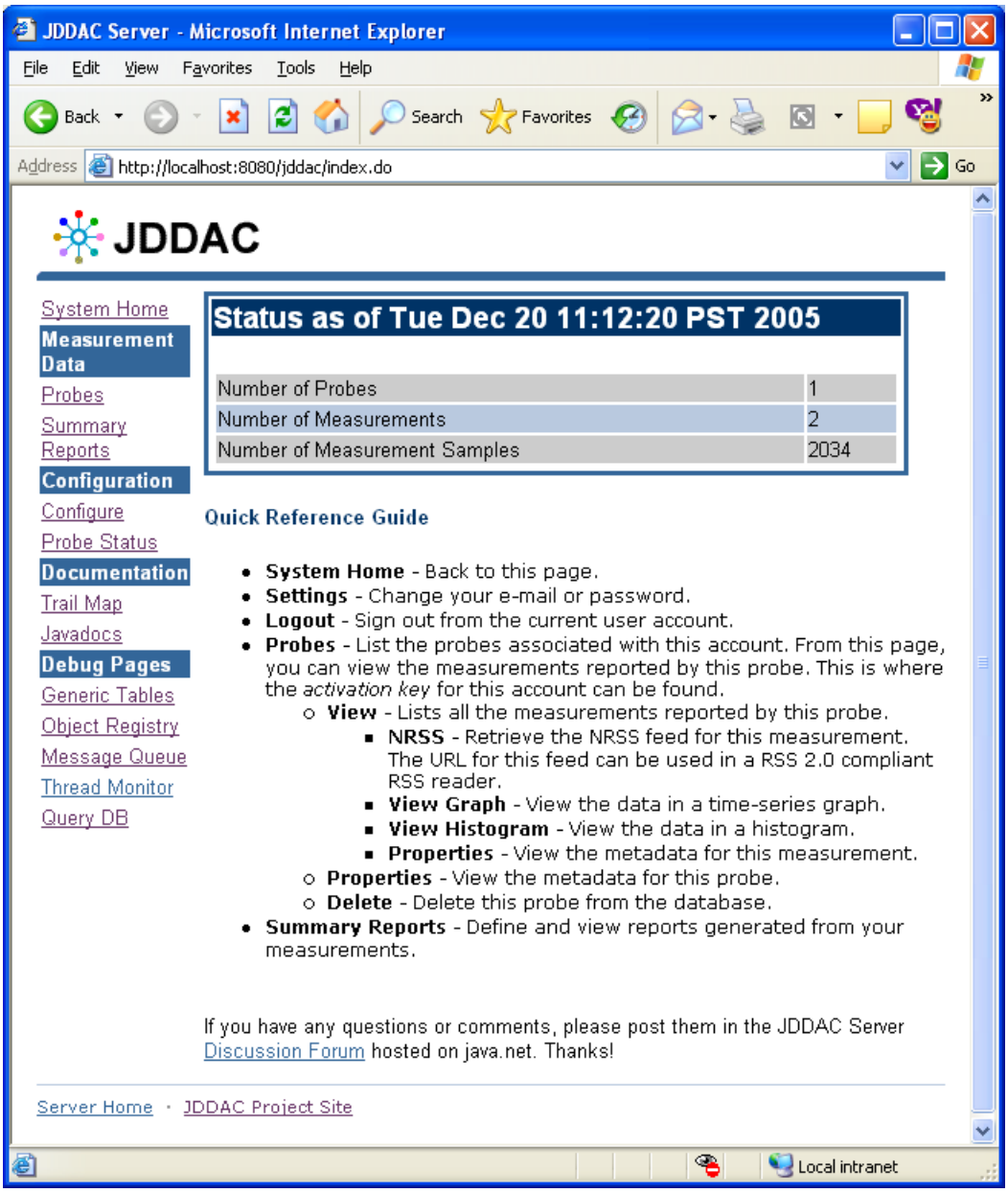


Figure 9-1 Server Home Page

By default, the system starts a probe named 'server' which makes two measurements, the amount of total memory and the amount of free memory in the server's JVM. You can see the list of probes known by the server by clicking on 'Probes'.

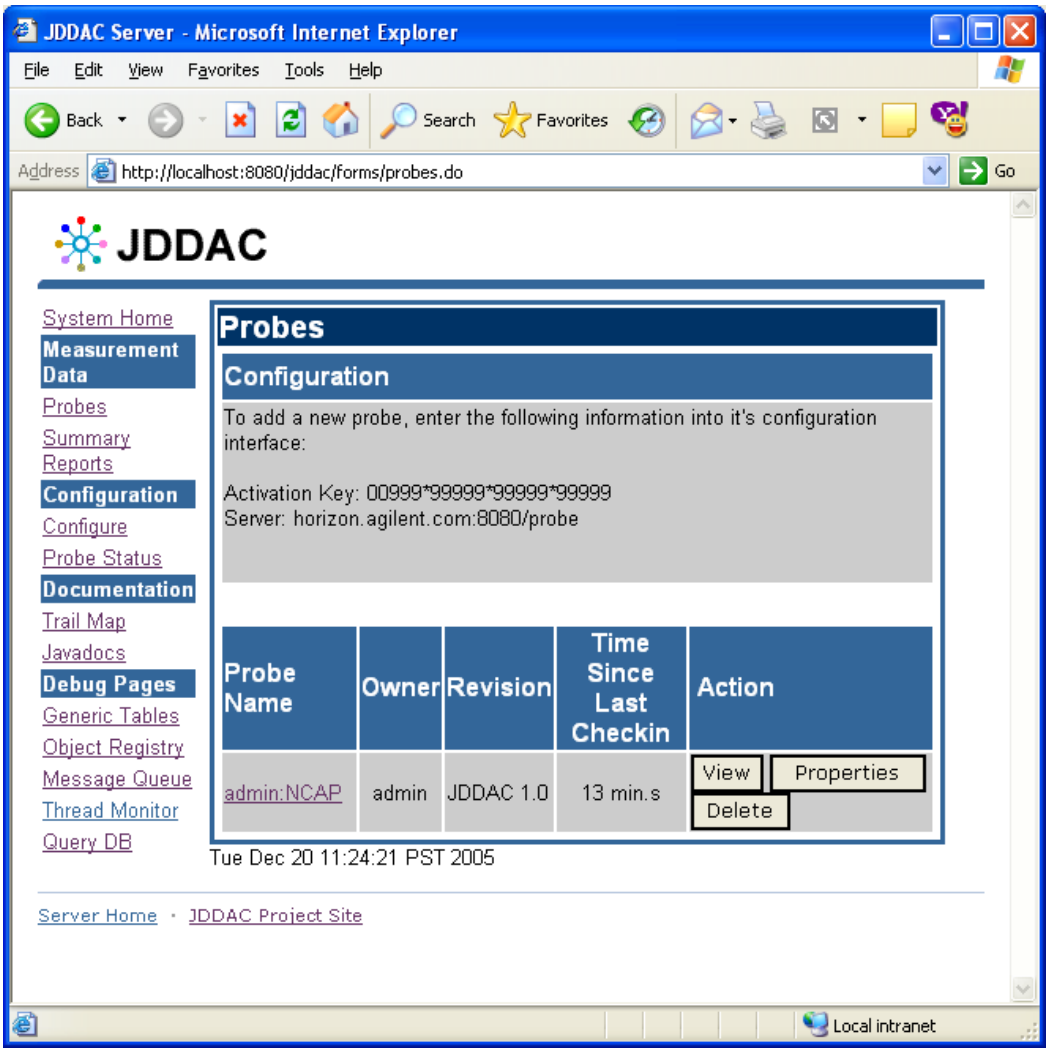


Figure 9-2 Probe Property Page

Clicking on the property button will show you the metadata associated with this probe. The metadata is presented in an XML format.

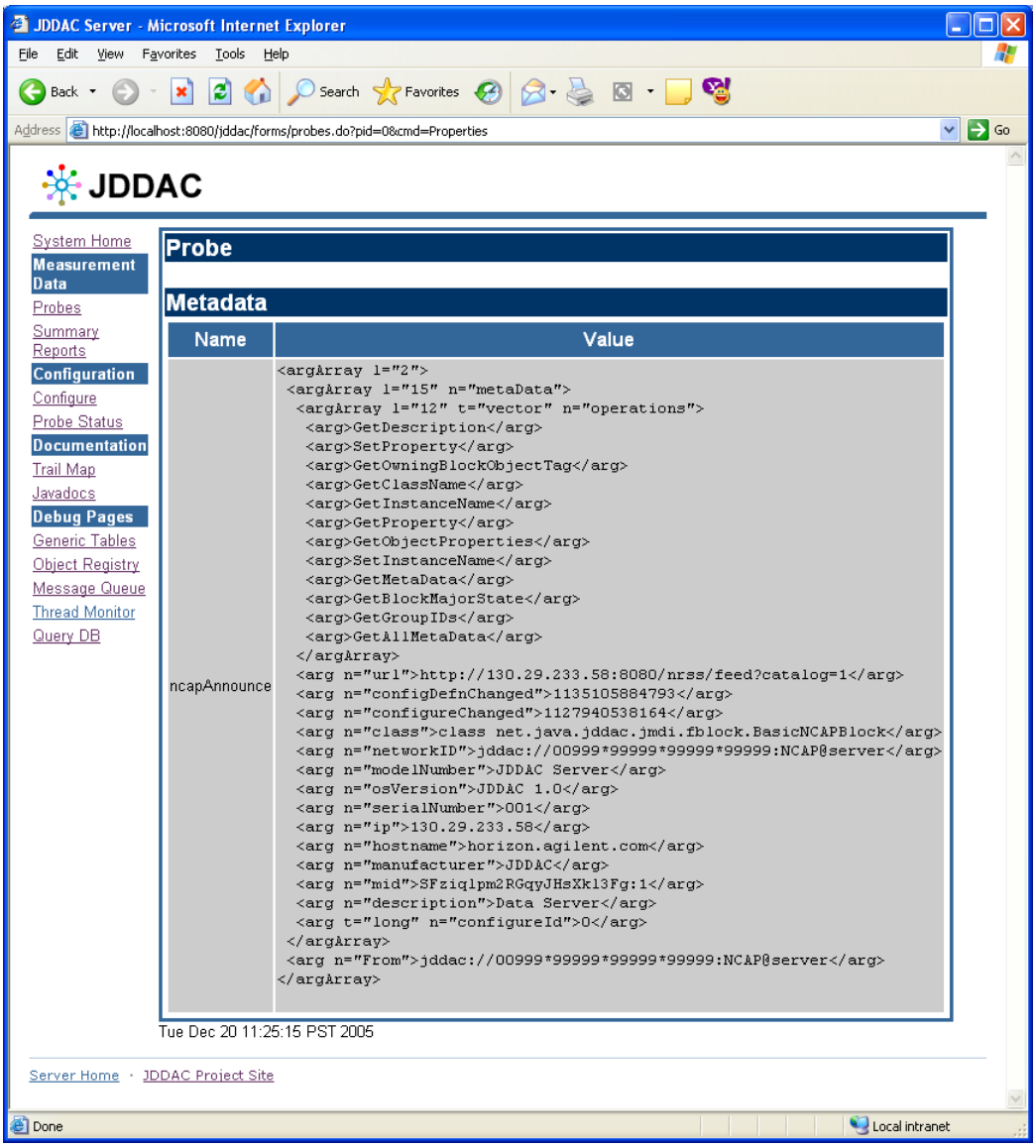


Figure 9-3 Probe Metadata Page

The View button brings up a list of measurements associated with this probe:

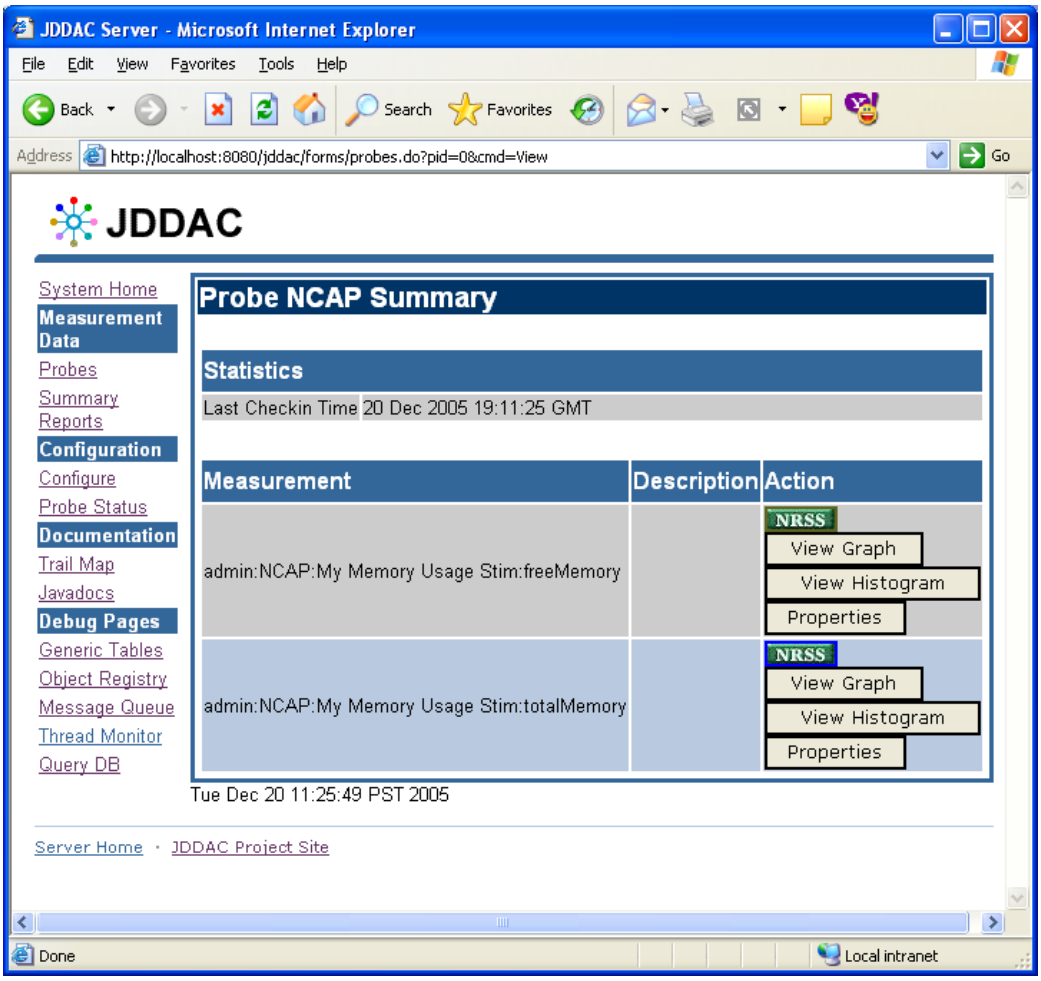


Figure 9-4 Measurement Listing Page

The Properties button brings up the metadata for this measurement, this time in a table format. This is metadata that was supplied by the probe when the probe first contacts the server. This metadata is usually derived from the Transducer Electronic Data Sheet (TEDS) associated with the transducer.

Figure 9-5 Measurement Metadata Page

The NRSS button brings up a NRSS feed for the measurement. This is a RSS 2.0 compatible feed which can be processed by RSS readers. The remaining buttons bring up either a graph or a histogram of the measurement. The time range to be graphed can be adjusted. To select an arbitrary start/stop date, choose the 'Arbitrary Time Range' selection under the Time Scale selection.

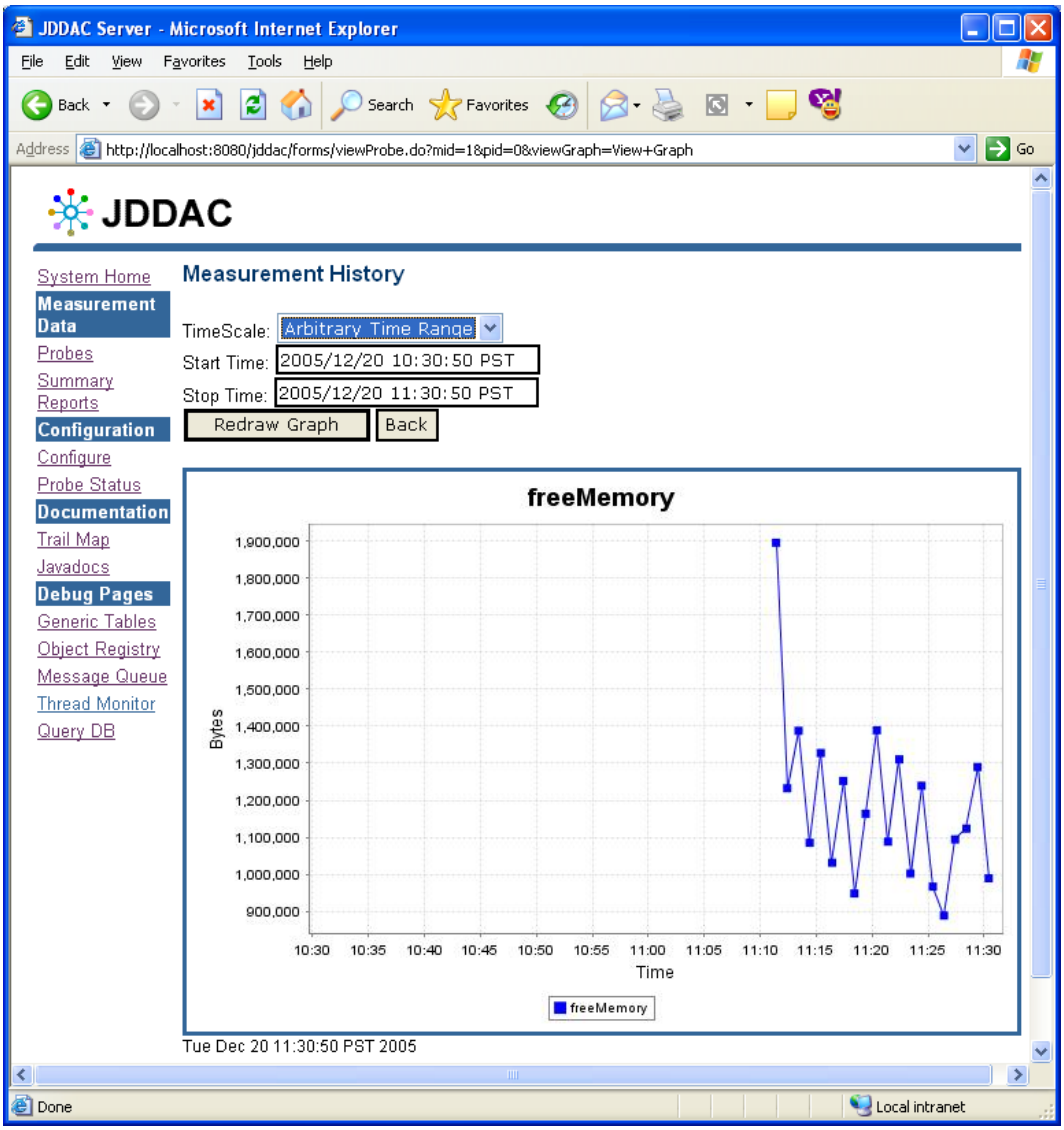


Figure 9-6 Measurement Graph Page

The Summary Reports option allow you to define and view specific reports:

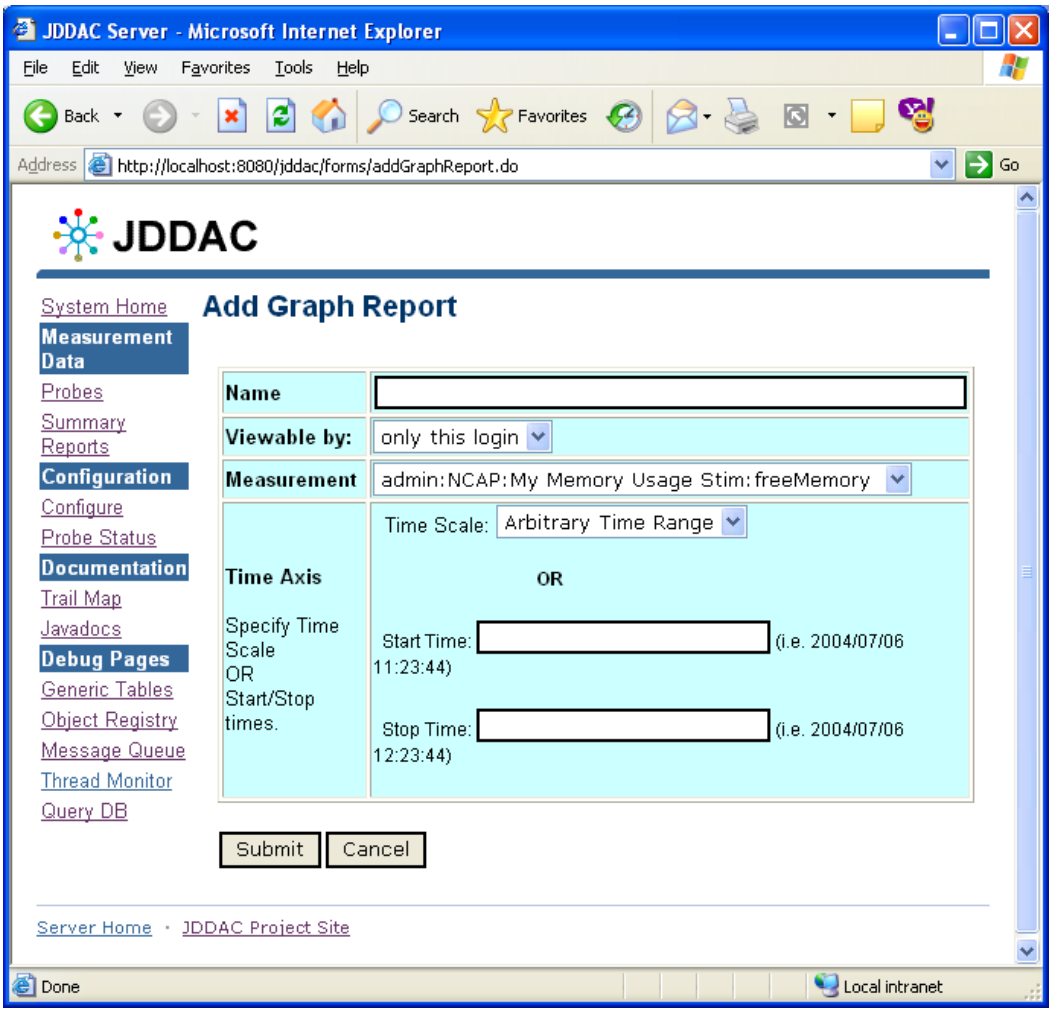


Figure 9-7 Define Graph Page

9.3 Probe Configuration

The configure page allows probes to be configured and new probe types to be added. Each probe belongs to a probe type, and each probe type has a particular configuration. This page allows you to edit the configurations of probe types, and thus the configurations that are sent to the probes.

Probe types are defined via ArgArray XML. You can use the Server XML as an example. There are

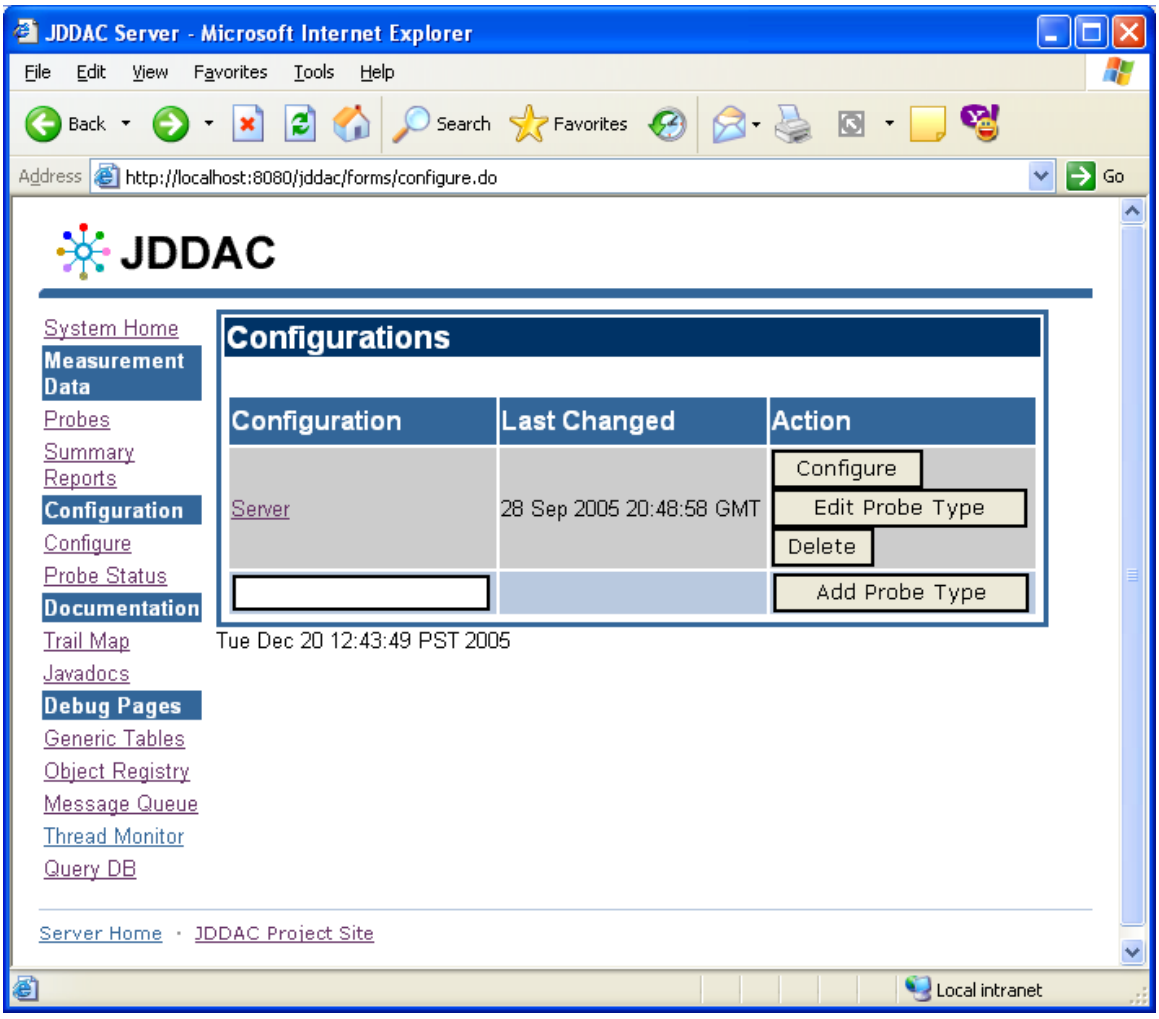


Figure 9-8 Probe Property Page

The probe status page lists the probes, their types, and whether or not themost recent configuration has been communicated to them.

9.4 Debug Pages

The debug pages section contains a number of diagnostic pages. These are generally used for server administration by revealing the internal server state for troubleshooting.

- **Generic Tables** shows the entities that are stored in the database and their associated metadata. It allow the user to edit the data stored in these tables.
- **ObjectRegistry** lists all the IEEE 1451 objects that currently exist in the system.
- **MessageQueue** lists the messages that are waiting to be delivered to their destinations, which is usually probes that report into the server.
- **Thread Monitor** lists the threads that are running in the system.

- **QueryDB** provides a mechanism for a user to issue SQL commands directly to the database.

10 Object Model

The JDDAC object model is based the object model from IEEE 1451.1. Application functionalities are contained within Block objects. The Block objects are connected via objects called Ports which are responsible for communication among the Block objects.

There are four kinds of Blocks: Function Blocks, commonly known as F-Blocks, Transducer Blocks, commonly known as T-Blocks, Transducer Interface Module, known as a TIM, and NCAP Blocks (no nick name).

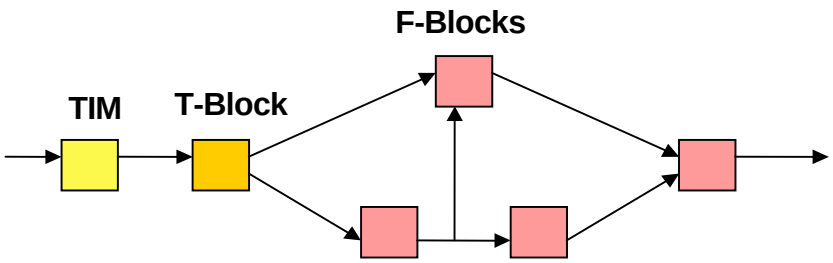
An F-Block is the most common type of Block. It typically contains some part of the application functionality. These are some example functionalities that are contained within F-Blocks:

- Measurement averaging
- Thresholding and alarming
- Generating CSV data
- Generating a measurement graph

A T-Block is responsible for orchestrating the measurement acquisition process, such as controlling data acqusition on a sampling frequency. It typically is connected to one or more TIM to acquire the data

A TIM is the software interface to the hardware. It encapsulates the logic necessary to use the drivers to acquire the data from the measurement hardware. TIM is where measurement data enters a JDDAC application or system. A TIM also has a Transducer Electronic data Sheet (TEDS) associated with it which describes the sensor that the TIM instance is associated with.

Lastly, the NCAP Block is the manager for a node, either a probe or a server. It is a singleton object. It contains status information about the Blocks in the system. It is also where messages addressed to the Blocks in a node is first sent.



A JDDAC application then is composed of a collection of Blocks wired together to form one or more measurement processing dataflows. Measurements enter the system through TIM modules and are then transformed as they flow from Block to Block.

The wiring among the Blocks in a JDDAC application is dynamic and changeable at run-time. This dynamic rewiring capabbility is what provides JDDAC systems with much of its flexibility.

11 Data Model

IEEE 1451.1 defines a loosely typed type system with essentially two entities: Argument and Arguemtn Array. An argument is a piece of data. As defined in 1451.1, it is essentially an IDL union type. In Java, the closest mapping would be java.lang.Object. An argument can be a simple type such as Integer or Float, or a more complex type such as Location or Time Representation.

An ArrgumentArray is essentially a collection of Arguments. In 1451.1, an ArgumentArray is an ordered collection of Arguments where the Arguments are referred to by their index. In JDDAC, that notion is extended to support name/value pairs. Thus, in JDDAC, an ArgumentArray takes on the additional responsibilities of an associative array. Each element in the array is associated with a string known as the index. This index is needed to retrieve the value from the Argument Array. Only one value can be associated with an index.

An Argument Array is also an Argument. Thus, an Argument Array can contain other Argument Arrays.

ArgumentArrays are the lingua franca in 1451.1, as well as in JDDAC. Most transactions and operations involve ArgArrays. And most data entities, rather than being specifically typed classes, are also ArgArrays. for instance, measurements and metadata are argarrays. A message between two Blocks is an argarray. Startup configuration is an argarray. A publication involves publishing an argarray, and the dynamic invocation mechanism takes an argarray as an argument and returns an argarray.

The ArgArray class has a number of helper methods to make it easy to insert and retrieve values. It is also easy to serialize argarray into XML as well as parse XML into argarray.

12 Measurement Model

As mentioned in the data model section, a measurement data sample in JDDAC is represented as an argarray. That is to say, a measurement is modeled as a collection of attributes.

These attributes can have any arbitrary names. The application designer is free to add any type of attribute to the measurement. There are a set of well-known attribute names defined in the MeasAttr class. These contain names such as:

- VALUE - value of measurement
- UNITS - units that measurement is represented in
- TIMESTAMP- timestamp of when measurement was made

and so forth. The default JDDAC components know how to interact with these names. for instance, the graphing Block knows to extract the UNITS field of a measurement when generating graphs. So when possible, use the built-in names.

A Measurement is made up a collection of measurement data samples from the same probe. A measurement will also have metadata that apply to all the measurement data samples that it contains. Rather than tagging each measurement data sample with all this metadata, these common attributes are collected together and associated with the measurement as a whole.

Related measurements can also be grouped into a Record.

13 Probe Model

The probe's attributes is essentially the metadata of its NCAP.

A probe may belong to one or more probe type. A probe type is essentially a collection of metadata attributes. A probe belongs to a probe type if its metadata has the same attributes as the collection of metadata defined in the probe type.

Probe type is important because it affects how probes are configured. In JDDAC, probes are not configured on a probe by probe basis, but rather on probe types. A particular configuration is tied to a probe type and affects all the probes in that probe type.

Thus in essence the probe type is a grouping mechanism that groups together probes that share common attributes. And all probes in a probe type share the same configuration. Note that probe type definitions can be very restrictive or lax. So a probe type can be defined in such a way that it affects only one probe.

14 Communication Model

There are two primary modes of communication in JDDAC: publish-subscribe and client server.

14.1 Publish-Subscribe

Within a JDDAc application (either probe or server), F-Blocks are connected via Port objects. In particular, Publisher Ports and Subscriber Ports, which implement a pub-sub infrastructure. A publisher publishes content on a topic, and subscribers which subscribe to that topic receives the content. In this manner, you can have one publisher publishes to many subscribers, many publishers publishing to one subscriber, or many publshers publishing to many subscribers. the publishers and subscribers are decoupled and does not know the identities of their counterparts. In fact, since publishers and subscribers can come and go dynamically, the identities may change during the life time of an application.

For more information on Pub-Sub ports, see IEEE 1451.1 specifiations section...

The pub-sub infrastructure for each node is implemented by a communcation fabric object which is responsible for delivering messages to the approproate recipients.

For inter-node communication, an F-Block called the Reporter comes into play. A reporter subscribes to topics as other Blocks. however, it's job is not to publish the content within the node, but to deliver the content to another node on the network. For a Reporter on the probe, the destination is typically a server.

The content is encoded in XML and delivered to the destination via HTTP. The response may be XML messages destined for the client. the reporter receives the HTTP reply, decodes the XML, and publishes the contents into the communication fabric to be picked up by subscriber ports within the node.

Currently, J2ME and J2SE applications do not listen for incoming messages. They only receive incoming messages in the response content of a HTTP reply.

On the J2EE server, a servlet is used to receive incoming messages via HTTP. So J2EE servers can both receive messages as well as generate requests.

14.2 Client-Server

In JDDAC, per 1451.1 specification, each Function Block implements a method called perform(). This method takes an operation ID, incoming arguments, and outgoing arguments. It's a mechanism for performing dynamic run-time dispatching.

For more information on client-server interaction, see IEEE 1451.1 specifiations section...

14.3 Messages

There are three types of messages that go on the wire: publications, client messages, and server messages. Client-server messages usually occur in pairs.

14.4 Transports

Before a message, in the form of a Java ArgArray object, goes on the wire, it needs to be encoded in XML and possibly compressed or encrypted. This is performed by the communication transport mechanism. The communication transport is composed of a number of coders. Each coder performs a different function, such as encoding, decoding, compression, or decompression. The coders are arranged in a pipeline fashion, with the output of one coder going into the input of another.

15 Naming Model

15.1 Overview

This document describes the Jddac ID usage and syntax. The Jddac Id is serves two primary functions:

1) Unique identifier for an entity (instance) or data element in the system corresponding to a 1451.1 ObjectID.
2) Provide a way to address and enable message delivery to a network addressable entity. This corresponds to the concept of an IEEE 1451.1 ObjectDispatchAddress.

Since the Jddac Id is used for message delivery, it can be treated as either a 'source identifier or a 'destination address' depending on the context. In most cases it will serve as a source identifier providing identity information to data publications but it also can be used as a destination address for directed publications or client/server communications.

The Jddac URI can be summarized with the following components:

```
Jddac://<identity>/<local path>?<additional parameters>
<identity> = <id>@<identityHost>
<local path> = <type>[/<name>]
```

Destination: jddac://<id>@<identityHost>/p/<topic> (for publications only)

Destination: jddac://<id>@<identityHost>/i/<entityName> (for client-server only)

From: jddac://<id>@<identityHost>/i/<entity name>

The <id>@<identityHost> portion is a globally unique form of identity where <id> is a unique string within the domain of <identityHost>. This combination is primarily used to uniquely identify a specific NCAP but can also be used to identify a specific user.

The following are example Jddac Ids.

User account 'glenne' on server jddac.labs.agilent.com:

jddac://glenn@jddac.labs.agilent.com

A block instance named 'reporter' in an ncap associated with the id ABCD-EFGH on the identity server jddac.labs.agilent.com:

jddac://ABCD-EFGH@jddac.labs.agilent.com/i/reporter

A measurement on the 'temperature' channel of the 'sensor' block instance:

jddac://ABCD-EFGH@jddac.labs.agilent.com/i/sensor?ch=temperature

15.2 Jddac URI and RFC2396

The Jddac URI is based on the URI syntax defined in RFC2396. Please refer to RFC2396 for complete definitions and character restrictions for these components. A portion is repeated here for easy reference. In all cases the Jddac URI is a subset of the RFC specification so a general URI parser based on the RFC can be used on Jddac URIs.

absoluteURI = scheme ":" (hier_part | opaque_part)
hier_part = (net_path | abs_path) ["?" query]
net_path = "//" authority [abs_path]
abs_path = "/" path_segments
authority = server
server = userinfo “@” hostport
hostport = host [":" port]
host = hostname | IPv4address
scheme = “jddac”

Jddac does not support registry based naming so the authority field does not include the option described in RFC2396 allowing authority to also be specified as <reg_name>.

path_segments = entity_instance | publication
entity_instance = “i/” name
publication = “p/” topic
name = 1*pchar
topic = 1*pchar

15.3 Implementation

The implementation of the Jddac Id is contained in the JddacId.java class.

16 Thread Model

The thread model describes how the threads traverse the network of blocks. JDDAC allows multiple threads to be active within the dataflow network, although in general, thread swapping is minimized to increase the performance.

Since the configuration model describes how the components, as well as the topology of the network organizes Block instances in a graph, the implementer of the blocks need to be cognizant of certain design patters which lead to deadlock.

Deadlock can occur when communication flows from block to block in a unique pattern that varies from thread to thread. For example, consider the diagram below that shows a message flowing from A to B to C and another flowing from C to B to D. If each block synchronizes to itself while processing the message then a deadlock will occur if A and B are locked by thread X while thread Y has locked C but has not yet locked B.

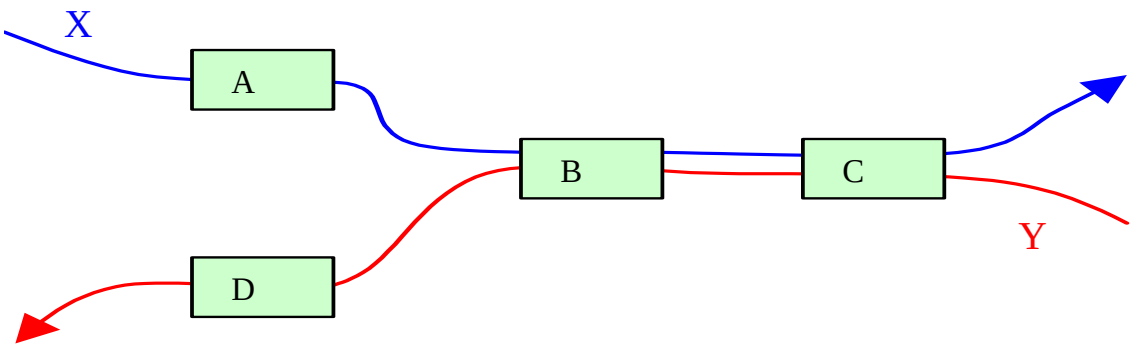


Figure 5. Remote Communication Port API

To address this problem blocks are required to treat any subscriptions they receive as immutable objects and to relinquish any internal synchronization locks before initiating a publish operation. As a result, blocks are free to keep references to subscriptions they receive and can even include all or portions of subscriptions in publications they generate.

However, any new state placed into a publication must subsequently be considered immutable. This usually means that new elements placed into a publication need to be copies of internal state rather than references to internal state. One approach is to simply do a deep copy on the entire message and then make changes. This however is inefficient if the message is large and only a small portion is being changed. This can optionally be addressed by only making copies of container objects above the new values.

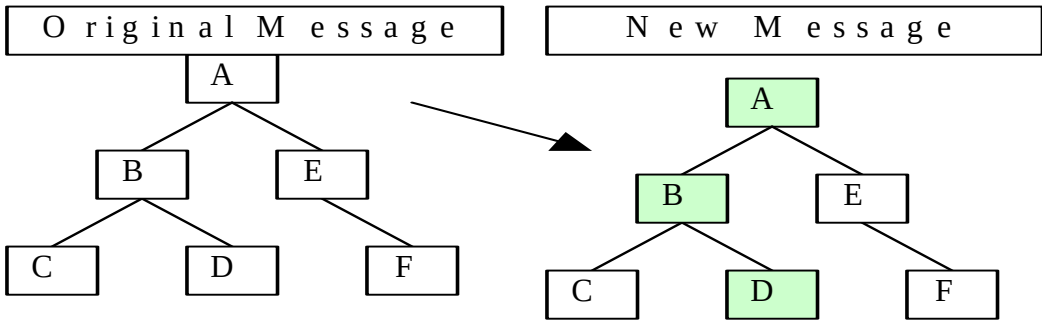


Figure 6. Remote Communication Port API

As an example, consider the figure above representing a container relationship between objects. If object D is replaced in the original message then container objects A and B must be made into copies before the message can be published from the Block. Example pseudo code showing these concepts is provided below. The key aspect to note is that the publication list must be cloned (and any necessary cloning in the message) before they synchronized region is exited.

```
publish(message) {
           // clone the subscriber list and
           // then publish the messages
           synchronize (this) {
               tmpSubList = subList.clone();
           }

           // No locks held at this point
           foreach sub in tmpSubList {
               port.send(message);
           }

          someMethod() {
          synchronize (this) {
             notify(message);  // This is illegal:
                               // publishing while locked
          }
}

// example notify callback on a new message being received
notify(message) {
    if (Thread.holdsLock(this)) throw new Exception();
    if (need to modify message) {
         synchronize (this) {
             tmpMsg = message.deepclone();
             modify(tmpMsg);  // Modify the copy
         }
                                }

    // No locks held at this point
    publish(tmpMsg);
}
```

Figure 7. Remote Communication Port API

17 Probe-Server Interaction

Currently, the interactions between the probe and the server are pretty minimal and can easily be extended.

When the probe first starts up, it publishes an announcement to the server containing its metadata. This notifies the server of the probe's existence. whenever the probe has measurements, it publishes the data on a topic called 'storeMeasurement' to the server. Upon receipt of the measurement, the server checks to see if the metadata for the measurement is known. If not. a message queued to be returned to the probe asking for the metadata to be sent up.

Thus, there are only four types of messages between the server and the probe.

- The message from the probe announcing its presence.
- The message from the probe with measurement.
- The message from the server asking for the measurement metadata.
- The message from the probe with the details of the measurement metadata.

18 Configuring Applications with XML

The configuration XML defines what blocks are instantiated and what their attributes are. Different blocks have different configuration attributes. Need to check the Block documentation for specifics.

This process is easier to learn by example. Thus, we will go through two examples. We'll examine the configuration file for JDDAC probe and JDDAC server and examine what it means.

18.1 Probe

Here is the XML for the probe configuration. Note that the entire configuration itself is an ArgArray.

```
<?xml version="1.0"?>

<argArray t="vector">
    <!-- ============================================================ -->
    <!-- Set default logging levels.  Note this object is not transient -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="Entity.description">Logger Defaults</arg>
        <arg n="instanceName">Logger</arg>
        <arg n="className">net.java.jddac.common.util.LogInit</arg>
        <arg n="register">false</arg>
        <argArray n="defaults">
            <arg n="Log">warning</arg>
        </argArray>
    </argArray>

    <argArray>
        <arg n="className">net.java.jddac.jmdi.fblock.BasicNCAPBlock</arg>
        <arg n="instanceName">NCAP</arg>
        <arg
n="NCAPBlock.xprobeNetworkId">jddac://99999*99999*99999*99999@server</arg>
        <argArray n="NCAPBlock.registry">
            <arg n="description">NCAP Block</arg>
            <arg n="manufacturer">Foobar Industries</arg>
            <arg n="modelNumber">JddacClient 1.0</arg>
            <arg n="serialNumber"></arg>
            <arg n="description">JDDAC Client Demo</arg>
            <arg n="osVersion">JDDAC 1.0</arg>
        </argArray>
        <argArray n="FunctionBlock.pubPorts">
            <argArray n="announce">
                <arg n="topic">ncapAnnounce</arg>
            </argArray>
        </argArray>
    </argArray>

    <!-- ============================================================ -->
    <!-- Configure handlers for the reporter encoder                  -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="className">net.java.jddac.jmdi.comm.DefaultCoder</arg>
```

```
        <arg n="instanceName">coder</arg>
        <arg n="DefaultCoder.defaultContentEncoding">mp</arg>
        <argArray n="DefaultCoder.coders">
            <arg n="mp">net.java.jddac.jmdi.comm.MultiPartCoder</arg>
        </argArray>
    </argArray>
    <argArray>
        <arg n="className">net.java.jddac.jmdi.comm.HttpTransport</arg>
        <arg n="instanceName">transport</arg>
        <arg n="HttpTransport.coder">coder</arg>
    </argArray>

    <!-- ============================================================ -->
    <!-- Configure the reporter to communicate with the server        -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="className">net.java.jddac.jmdi.fblock.Reporter</arg>
        <arg n="instanceName">Reporter</arg>
        <arg t="long" n="Reporter.connectInterval">10</arg>
        <arg n="Reporter.transport">transport</arg>
        <argArray n="FunctionBlock.subPorts">
            <argArray n="incomingData">
                <argArray n="qual" t="vector">
                    <arg>storeMeasurement</arg>
                    <arg>ncapAnnounce</arg>
                </argArray>
            </argArray>
            <argArray n="queueSerialize">
                <arg n="qual">destroyApp</arg>
            </argArray>
        </argArray>
        <arg n="Reporter.timeout">300s</arg>
    </argArray>

    <!-- ============================================================ -->
    <!-- Configure a Block to echo publications to stdout             -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="className">net.java.jddac.jmdi.fblock.EchoBlock</arg>
        <arg n="instanceName">EchoBlock</arg>
        <argArray n="FunctionBlock.subPorts">
            <argArray n="incomingData">
                <arg n="qual">storeMeasurement</arg>
            </argArray>
        </argArray>
    </argArray>

    <!-- ============================================================ -->
    <!-- Configure a TIM to monitor JVM memory utilization            -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="instanceName">My Memory Usage Stim</arg>
        <arg n="className">net.java.jddac.jmdi.transducer.MemoryUsage</arg>
    </argArray>

    <argArray>
        <arg n="instanceName">My Memory Usage TBlock</arg>
        <arg
n="className">net.java.jddac.jmdi.fblock.BasicTransducerBlock</arg>
        <arg n="Entity.description">Demo Basic Transducer Block for Memory
Usage</arg>
        <arg n="BasicTransducerBlock.transducerName">My Memory Usage
Stim</arg>
        <arg t="int" n="BasicTransducerBlock.period">10</arg>
        <argArray n="FunctionBlock.pubPorts">
            <argArray n="result">
                <arg n="topic">storeMeasurement</arg>
            </argArray>
```

```
        </argArray>
    </argArray>

    <!-- ============================================================ -->
    <!-- trigger an announce to the world we are now up and running   -->
    <!-- ============================================================ -->
    <argArray>
        <arg n="instanceName">NCAP</arg>
        <arg n="NCAPBlock.announceNCAP">true</arg>
    </argArray>
</argArray
```

Each ArgArray within the outer most ArgArray defines a Block to be instantiated. In the probe XML here, there are eight such ArgArray fragments, meaning that eight Blocks are instantiated in the probe application.

The <arg> and <argArray> tags within each Block's ArgArray XML fragment defines the attributes for the Block. There are some common attributes which needs to be in every Block XML fragment:

- instanceName – Specifies a human readable name which will be used to refer to this Block. Instance names should be unique within a JDDAC application.

- className – Specifies the fully qualified Java class name of the object.

The rest of the attributes varies depending on the class being instantiated.

In most instances, a JDDAC probe application will have:

- A BasicNCAPBlock. This Block is the manager block which contains the metadata for the application and a registry of all the Objects in the system.

- A Reporter. This Block is responsible for communicating with the server.

- A derived class of BasicTIM. This Block encapsulates the logic required to interface with a specific transducer. There may be multiple instances of this class to interface with different instances of transducers.

- A BasicTransducerBlock. This Block is associated with a BasicTIM. There may be multiple instances of this class to interface with different instances of basicTIM objects. A BasicTransducerBlock is responsible for responding to triggers to read the data from the TIM.

Blocks are connected to one another via Publisher Ports and Subscription Ports. In this context, being connected means that one block is able to send measurements to another block to be processed.

A Block publishes measurement data onto the communication fabric using Publisher Ports. A Publisher Port publishes data into the communication fabric using a particular topic. A Block can have many Publisher Ports. They are defined in the configuration XML as follows. This example is taken from the Reporter:

```
<argArray n="FunctionBlock.pubPorts">
    <argArray n="result">
        <arg n="topic">storeMeasurement</arg>
    </argArray>
</argArray>
```

This XML fragment defines a Publisher Port called “result”. The name “result” is used internally within the Block and has no bearings on the publication topic. The topic that it is publishing on is called “storeMeasurement”.

A Block receives measurements from the communication fabric using Subscriber Ports. Like Publisher Ports, Subscriber Ports are associated with a specific topic. When a publication is published on a topic, all Subscriber Ports listening on that topic receives a copy of the message.

Subscriber Ports are defined as follows:

```
<argArray n="FunctionBlock.subPorts">
    <argArray n="incomingData">
        <arg n="qual">storeMeasurement</arg>
    </argArray>
</argArray>
```

18.2 Server

Here is the XML for the server configuration.

```
<?xml version="1.0"?>

<argArray t="vector">

 <!--  NCAP Block  -->
  <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.BasicNCAPBlock</arg>
  <arg n="instanceName">NCAP</arg> <!-- Special name for ncap -->
  <arg n="Entity.description">NCAP Block</arg>
  <arg
n="NCAPBlock.probeNetworkId">jddac://00999*99999*99999*99999@server</arg>
  <argArray n="NCAPBlock.registry">
    <arg n="manufacturer">JDDAC</arg>
    <arg n="modelNumber">JDDAC Server</arg>
    <arg n="serialNumber">001</arg>
    <arg n="description">Data Server</arg>
    <arg n="osVersion">JDDAC 1.0</arg>
    <arg n="url">http://www.example.com:8080/server/probe</arg>
  </argArray>
  <argArray n="FunctionBlock.pubPorts">
    <argArray n="announce">
      <arg n="topic">serverAnnounce</arg>
    </argArray>
  </argArray>
 </argArray>

 <!-- Setup logging -->
 <argArray>
  <arg n="includeFile">../etc/startup/logging.xml</arg>
 </argArray>

 <!-- Setup database connection -->
```

```
 <argArray>
                                                        <arg
n="includeFile">../etc/startup/dbCon.hsqldb.xml</arg>
 </argArray>

 <!-- Setup basic server blocks -->
 <argArray>
  <arg n="includeFile">../etc/startup/basicBlocks.xml</arg>
 </argArray>

 <!-- Setup measurement error handling blocks -->
 <argArray>
  <arg n="includeFile">../etc/startup/errorSplitter.xml</arg>
 </argArray>

 <!-- Setup graph report blocks -->
 <argArray>
  <arg n="includeFile">../etc/startup/graphReport.xml</arg>
 </argArray>

 <!-- Setup CSV exrpot block -->
 <argArray>
  <arg n="includeFile">../etc/startup/csvExport.xml</arg>
 </argArray>

 <!-- Setup NRSS blocks -->
 <argArray>
  <arg n="includeFile">../etc/startup/nrss.xml</arg>
 </argArray>

 <!-- Setup federation -->
 <argArray>
  <arg n="includeFile">../etc/startup/federate.xml</arg>
 </argArray>

 <!-- Send servers NCAP announcement -->
 <argArray>
  <arg n="instanceName">NCAP</arg> <!-- Special name for ncap -->
  <arg n="NCAPBlock.announceNCAP">true</arg>
 </argArray>
</argArray
```

The Server configuration XML permits syntax to include other configuration files. This is because the Server configuration files are usually fairly large, and this allows different subsystems to be managed more easily.

This is the XML for the largest of the include files, the one containing the definition for the basic Blocks which make up the server.

```
<?xml version="1.0"?>

<argArray t="vector">

 <!-- User database access -->
 <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.UserBlock</arg>
  <arg n="instanceName">User</arg>
  <arg n="db">UserDb</arg>
 </argArray>

 <!-- Block to hole "backchannel" messages  -->
 <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.MessageQueueBlock</arg>
```

```
  <arg n="instanceName">MessageQueueBlock</arg>
  <arg n="Entity.description">Demo Message Queue Block</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <argArray n="MessageQueueBlock.commSubList" t="vector" l="1">
   <argArray>
    <!-- Normal, both delivered & not delivered -->
    <arg n="deliveryRestrictions" t="int">0</arg>
    <!-- Not local host -->
    <arg n="localHost" t="boolean">false</arg>
    <!-- Pubs & clients -->
    <!-- <arg n="type">p</arg> -->
    <!-- Any path -->
    <!-- <arg n="path">null</path> -->
   </argArray>
  </argArray>
  <argArray n="FunctionBlock.pubPorts">
    <argArray n="gettingMessagesEvent">
      <arg n="topic">gettingProbeMessages</arg>
    </argArray>
  </argArray>
</argArray>

<!-- Glue from Servlet to Block  -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.comm.ServerPimsCoder</arg>
 <arg n="instanceName">JddacCoder</arg>
 <argArray n="DefaultCoder.coders">
  <arg n="wbxml">net.java.jddac.jmdi.comm.xml.WbxmlObjectCoder</arg>
  <arg n="zip">net.java.jddac.jmdi.comm.crypto.ZipCoderJ2se</arg>
  <arg n="gzip">net.java.jddac.jmdi.comm.crypto.GZipCoderJ2se</arg>
  <arg n="mp">net.java.jddac.jmdi.comm.MultiPartCoder</arg>
 </argArray>
</argArray>
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.TransportBridgeBlock</arg>
 <arg n="instanceName">TransportBridgeBlock</arg>
 <arg n="Entity.description">Transport Bridge Block</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <arg n="transportBridgeBlock.msgQueue">MessageQueueBlock</arg>
</argArray>
<argArray>
 <arg n="className">net.java.jddac.jmdi.comm.HttpTransport</arg>
 <arg n="instanceName">JddacServlet</arg>
 <arg n="HttpTransport.coder">JddacCoder</arg>
 <arg n="HttpTransport.messageHandler">TransportBridgeBlock</arg>
 <argArray n="FunctionBlock.pubPorts">
   <argArray n="probeConnect">
           <arg n="topic">probeConnect</arg>
   </argArray>
 </argArray>
</argArray>

<!-- Probe Block -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.ProbeBlock</arg>
 <arg n="instanceName">ProbeBlock</arg>
 <arg n="Entity.description">Probe Block</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <arg n="db">DataDb</arg>
 <argArray n="FunctionBlock.subPorts">
   <argArray n="probeConnect">
     <arg n="qual">probeConnect</arg>
   </argArray>
   <argArray n="ncapMetadata">
     <argArray n="qual" t="vector">
       <arg>ncapAnnounce</arg>
       <arg>serverAnnounce</arg>
     </argArray>
```

```
    </argArray>
  </argArray>
</argArray>

<!-- Measurement info -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.MeasInfoBlock</arg>
 <arg n="instanceName">MeasInfoBlock</arg>
 <arg n="Entity.description">Measurement Info Block</arg>
 <arg n="db">DataDb</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <argArray n="FunctionBlock.subPorts">
   <argArray n="receiveMeasSub">
     <argArray n="qual" t="vector">
       <arg>storeMeasurement</arg>
       <arg>storeErrMeasurement</arg>
     </argArray>
   </argArray>
 </argArray>
 <argArray n="FunctionBlock.pubPorts">
   <argArray n="resendMeasPub">
           <arg n="topic">measWithMidDb</arg>
   </argArray>
 </argArray>
</argArray>

<!-- Metadata storage block -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.MetadataStoreBlock</arg>
 <arg n="instanceName">MetadataStoreBlock</arg>
 <arg n="Entity.description">Metadata Store Block</arg>
 <arg n="db">DataDb</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <argArray n="FunctionBlock.subPorts">
   <argArray n="metadata">
     <argArray n="qual" t="vector">
       <arg>metadata</arg>
       <arg>errMetadata</arg>
     </argArray>
   </argArray>
   <argArray n="gettingMessages">
     <arg n="qual">gettingProbeMessages</arg>
   </argArray>
   <argArray n="receiveMeasSub">
           <arg n="qual">measWithMidDb</arg>
   </argArray>
 </argArray>
 <argArray n="FunctionBlock.pubPorts">
   <argArray n="resendMeasPub">
           <arg n="topic">measWithMeta</arg>
   </argArray>
 </argArray>
</argArray>

<!-- Record storage block -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.RecordStoreBlock</arg>
 <arg n="instanceName">RecordStoreBlock</arg>
 <arg n="Entity.description">Record Store Block</arg>
 <arg n="db">DataDb</arg>
 <argArray n="FunctionBlock.subPorts">
   <argArray n="measIn">
     <arg n="qual">measWithMeta</arg>
   </argArray>
 </argArray>
 <argArray n="FunctionBlock.pubPorts">
   <argArray n="measOut">
     <arg n="topic">measWithAll</arg>
```

```
    </argArray>
  </argArray>
</argArray>

<!-- Report DB for getCount query -->
<argArray>
 <arg n="instanceName">measQueryDb</arg>
 <arg n="className">net.java.jddac.jmdi.fblock.ReportDbAccess</arg>
 <arg n="ReportDbAccess.rowSpec">count(*)</arg>
 <arg n="ReportDbAccess.mainTable">measurements</arg>
 <argArray n="ReportDbAccess.queries">
   <argArray n="getCount">
   </argArray>
         <argArray n="byMid">
           <arg n="where">id = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>mid</arg>
     </argArray>
         </argArray>
         <argArray n="byPid">
     <argArray n="tables" t="stringArray">
       <arg>meas</arg>
       <arg>meas.id = measurements.id</arg>
     </argArray>
           <arg n="where">meas.pid = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>pid</arg>
     </argArray>
         </argArray>
         <argArray n="getCountByMid">
           <arg n="where">id = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>mid</arg>
     </argArray>
         </argArray>
   <argArray n="getLastByMid">
           <arg n="rowSpec">value, time, lat, lon, alt</arg>
           <arg n="where">id = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>mid</arg>
     </argArray>
           <argArray n="preColumns">
     <arg n="hsqldb">limit 0 1</arg>
     </argArray>
     <argArray n="postWhere">
       <arg n="hsqldb">order by time desc</arg>
       <arg n="mysql">order by time desc limit 0,1</arg>
     </argArray>
   </argArray>
   <argArray n="getFirstTimeByMid">
           <arg n="rowSpec">time</arg>
           <arg n="where">id = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>mid</arg>
     </argArray>
           <argArray n="preColumns">
     <arg n="hsqldb">limit 0 1</arg>
     </argArray>
     <argArray n="postWhere">
       <arg n="hsqldb">order by time asc</arg>
       <arg n="mysql">order by time asc limit 0,1</arg>
     </argArray>
   </argArray>
   <argArray n="getLastTimeByMid">
           <arg n="rowSpec">time</arg>
           <arg n="where">id = ?</arg>
     <argArray n="sqlParms" t="stringArray">
       <arg>mid</arg>
```

```
      </argArray>
            <argArray n="preColumns">
        <arg n="hsqldb">limit 0 1</arg>
      </argArray>
      <argArray n="postWhere">
        <arg n="hsqldb">order by time desc</arg>
        <arg n="mysql">order by time desc limit 0,1</arg>
      </argArray>
    </argArray>
    <argArray n="getCurrentByMid">
            <arg n="rowSpec">value, time, lat, lon, alt</arg>
            <arg n="where">id = ?</arg>
      <argArray n="sqlParms" t="stringArray">
        <arg>mid</arg>
      </argArray>
            <argArray n="preColumns">
        <arg n="hsqldb">limit 0 1</arg>
      </argArray>
      <argArray n="postWhere">
        <arg n="hsqldb">order by time desc</arg>
        <arg n="mysql">order by time desc limit 0,1</arg>
      </argArray>
    </argArray>
    <argArray n="getLowByMid">
            <arg n="rowSpec">value, time, lat, lon, alt</arg>
            <arg n="where">id = ?</arg>
      <argArray n="sqlParms" t="stringArray">
        <arg>mid</arg>
      </argArray>
            <argArray n="preColumns">
        <arg n="hsqldb">limit 0 1</arg>
      </argArray>
      <argArray n="postWhere">
        <arg n="hsqldb">order by value asc</arg>
        <arg n="mysql">order by value asc limit 0,1</arg>
      </argArray>
    </argArray>
    <argArray n="getHighByMid">
            <arg n="rowSpec">value, time, lat, lon, alt</arg>
            <arg n="where">id = ?</arg>
      <argArray n="sqlParms" t="stringArray">
        <arg>mid</arg>
      </argArray>
            <argArray n="preColumns">
        <arg n="hsqldb">limit 0 1</arg>
      </argArray>
      <argArray n="postWhere">
        <arg n="hsqldb">order by value desc</arg>
        <arg n="mysql">order by value desc limit 0,1</arg>
      </argArray>
    </argArray>
          <argArray n="getMark">
      <arg n="rowSpec">id</arg>
      <arg n="mainTable">record</arg>
            <argArray n="preColumns">
        <arg n="hsqldb">limit 0 1</arg>
      </argArray>
      <argArray n="postWhere">
        <arg n="hsqldb">order by id desc</arg>
        <arg n="mysql">order by id desc limit 0,1</arg>
      </argArray>
          </argArray>
  </argArray>
</argArray>

<!-- RDBMS storage of measurement data -->
<argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.MeasStoreBlock</arg>
```

```
  <arg n="instanceName">MeasStoreBlock</arg>
  <arg n="Entity.description">Measurement Store Block</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="MeasStoreBlock.dbBlock">DataDb</arg>
  <arg n="MeasStoreBlock.insertSql">insert into measurements (id, value,
          time, lat, lon, alt, record_id, metadata_id) values (?, ?, ?, ?,
          ?, ?, ?, ?) </arg>
  <arg n="MeasStoreBlock.deleteSql">delete from measurements where id =
          ?</arg>
  <arg n="MeasStoreBlock.reportDb">measQueryDb</arg>
  <argArray n="MeasStoreBlock.insertCondition" t="vector">
    <argArray>
      <arg n="type">argExtract</arg>
      <argArray n="fields" t="vector">
        <arg>value</arg>
      </argArray>
            <arg n="exists">true</arg>
    </argArray>
  </argArray>
  <argArray n="MeasStoreBlock.parameterList" t="vector">
    <arg>middb</arg>
    <arg>value</arg>
    <argArray>
     <arg n="name">timestamp</arg>
     <arg n="type">time</arg>
    </argArray>
    <argArray>
     <arg n="name">location</arg>
     <arg n="insertAltitude" t="boolean">true</arg>
     <arg n="type">location</arg>
    </argArray>
    <arg>rid</arg>
    <arg>metaId</arg>
  </argArray>
  <argArray n="FunctionBlock.subPorts">
    <argArray n="measurements">
      <arg n="qual">measWithAll</arg>
    </argArray>
  </argArray>
</argArray>

<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.ConfigureBlock</arg>
 <arg n="instanceName">ConfigureBlock</arg>
 <arg n="Entity.description">Configuration Block</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <arg n="db">DataDb</arg>
 <argArray n="FunctionBlock.subPorts">
   <argArray n="ncapAnnounce">
     <argArray n="qual" t="vector">
       <arg>ncapAnnounce</arg>
       <arg>serverAnnounce</arg>
     </argArray>
   </argArray>
   <argArray n="gettingMessages">
     <arg n="qual">gettingProbeMessages</arg>
   </argArray>
 </argArray>
</argArray>

<!-- JFreeChart plot of measurement data -->
<argArray>
 <arg n="className">net.java.jddac.jmdi.fblock.JFreeChartBlock</arg>
 <arg n="instanceName">JFreeChartBlock</arg>
 <arg n="Entity.description">JFreeChart Graph Block</arg>
 <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 <arg n="JFreeChartBlock.dbBlock">DataDb</arg>
</argArray>
```

```
 <argArray>
  <arg n="className">net.java.jddac.jmdi.fblock.ReportStoreBlock</arg>
  <arg n="instanceName">ReportStoreBlock</arg>
  <arg n="Entity.description">Report Store Block</arg>
  <arg n="db">DataDb</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
 </argArray>

</argArray>
```

19 Dynamic Configuration with Server GUI

This chapter shows how to configure probes from the server.

there are several steps:

1. Define a probe type. This only needs to be done once.

2. Go to the configure page for that probe type and put in the actual values.

The first process involves defining an XML file which defines the criteria for a probe to belong to this probe type, and maps the attributes in the probe configuration to GUI elements.

the second page involves setting the actual values.

20 Database Storage

21 NRSS Data Feed

22 CSV Data Feed

22.1 Overview

This document describes the usage of the CSV data feed. This is an http request with parameters that allow specifying what data is to be returned in a CSV formatted file.

22.2 Request Parameters

The following table describes the parameters that can be used on the data feed request. The host and path portion of the URL should be obtained from the data/representation/url element of the NRSS feed.

Parameter Name	**Default Value**	**Description**
mid	None	The measurement database ID. This number can be found in the measurement links on the probe page. Either a mid or group parameter is required to be specified.
group	None	The organizational group. Setting this to tsd (the default) and not specifying a mid will get all trial measurements. Set this to vdnl to get all Vodaphone trail measurements. Either a mid or group parameter is required to be specified.
startTime	If neither startTime nor startMark is specified, then the default startTime will be 24 hours before the request time.	The time of the first measurement to return. Should be formatted as yyyy-MM-dd HH:mm:ss.SSS or yyyy-MM-dd HH:mm:ss.SSS ZZZ and defaults to GMT.
stopTime	None.	The time of the last measurement to return. Should be formatted as yyyy-MM-dd HH:mm:ss.SSS or yyyy-MM-dd HH:mm:ss.SSS ZZZ and defaults to GMT.

startMark	None.	The lowest possible mark value of measurements to return.
stopMark	None.	The highest possible mark value of measurements to return.

22.3 Data Returned

The returned CSV file has a title row followed by any matching measurements. The columns of the csv file are:

Column #	Column Title	Description
1	Id	The ID of the measurement. The JddacId form is described here: http://cm.labs.agilent.com/SSLDocs/jddac/architecture/JddacId.htm. A typical measurement ID will look like: jddac://00544*98987*38739*44353:M4@hpie12.labs.agilent.com/i/PhoneInfoTIM?ch=signalQuality.
2	time	The time the measurement was made. This will be formated as: yyyy-MM-dd HH:mm:ss.SSS and defaults to GMT.
3	lat	The latitude of the location the measurement was made in degrees.
4	lon	The longitude of the location the measurement was made in degrees.
5	value	The value of the measurement.
6	mark	The mark number for this measurement. Measurements with the same mark number were scheduled to take place at the same time. The mark number is always increasing, based on when the measurement was received by the server.

22.4 Incremental Measurement Retrieval

It is often desirable to access any new measurements that haven't been downloaded previously. The following algorithm can be used to retrieve any new measurements received by the server since the last request. As shown in the

algorithm, to guaranty all measurements are returned the stop mark number must be read from the NRSS feed.

```
startMarkNumber=0
while(true)
{
    endMarkNumber=getNrssMarkValue();
    dataChunk=getDataRange(startMarkNumber, stopMarkNumber);
    processData(dataChunk);
    startMarkNumber=endMarkNumber
}
```

22.5 Example

URL Request:

http://hpie12.labs.agilent.com/rome/nrss/data/csv?group=vdnl&startTime=2005-05-19%2020:00:00.000&stopTime=2005-05-20%2002:00:00.000

Data Returned

```
"id”,"time","lat","lon","value"
jddac://00:joe@hpie12.labs.agilent.com/i/PhoneInfoTIM?ch=BSIC,2005-05-19
20:11:39.000,0.0,0.0,54.0,2345
jddac://00:joe@hpie12.labs.agilent.com/i/PhoneInfoTIM?ch=Service_City,2005-05-19
20:11:39.000,0.0,0.0,0.0,2345
jddac://00:joe@hpie12.labs.agilent.com/i/PhoneInfoTIM?ch=Service_SuppressRoaming,20
05-05-19 20:11:39.000,0.0,0.0,0.0,2345
```

23 JDDAC web services Interface

The JDDAC web services interface is based on using HTTP GET and POST requests to send commands to the server and receiving an XML doucment as the reply. It permits a user to manage a JDDAC system using HTTP and XML. This chapter describes the capabilities offered by the JDDAC web services (JDDAC-ws) interface.

23.1 Introduction

IEEE 1451.1 Function Blocks support a perform() method where method dispatching takes place dynamically at run time. JDDAC-ws allows the perform() method of the Function Blocks in the system to be invoked via HTTP.

By default, the JDDAC-ws URL is at the following URL

/rest

It takes the following parameters:

- object - This is the instance name of the destination Function Block (required).
- method - This is the method name of the destination Function Block. Note that this is not necessarily the method name yielded from the java introspection process, but a string name defined within the specific Function Block subclass definitions. (required).
- p0 - First parameter. (optional)
- p1 - Second parameter. (optional)
- ...
- p9 - Tenth parameter. (optional)

The parameters must be provided in order (i.e. having p0 and p2 but not p1 is not allowed).

The HTTP transaction can be either GET or POST.

The return document from the transaction is a document containing an ArgumentArray encoded in XML format.

The following sections contain documentation for the API. The API is divided into three sections: server, probes, and measurement.

23.2 Server

The APIs in this section deal with the status of the server itself.

1. ProbeCount - Returns number of known probes
   - object -
   - method -
2. MeasurementCount - Returns number of known measurements
   - object -
   - method -
3. MeasurementSampleCount - Returns number of measurements data samples.
   - object -
   - method -
4. Shutdown - Shuts down server.
   - object -
   - method -

23.3 Probes

The APIs in this section deal with the probes served by the server.

1. GetAllProbes - Returns list of all probes.
   - object -
   - method -
2. GetProbeAttr - Returns list of attributes for specific probe.
   - object -
   - method -
   - p0 - ID of probe

3. GetProbeMeas- Returns list of measurements for specific probe.
    - object -
    - method -
    - p0 - ID of probe
4. DeleteProbes - Deletes a specific probe.
    - object -
    - method -
    - p0 - ID of probe
5. GetAllConfigStatus - Returns configuration status for all probes.
    - object -
    - method -
6. GetConfigStatus - Returns configuration status for a single probe.
    - object -
    - method -
    - p0 - ID of probe
7. UploadProbeType - Uploads a new probe type XML into server.
    - object -
    - method -
    - p0 - XML document defining the probe type.

23.4 Measurements

The APIs in this section deal with the measurements stored by the server.

1. GetAllMeasurements - Returns list of all measurements.
    - object -

- method -

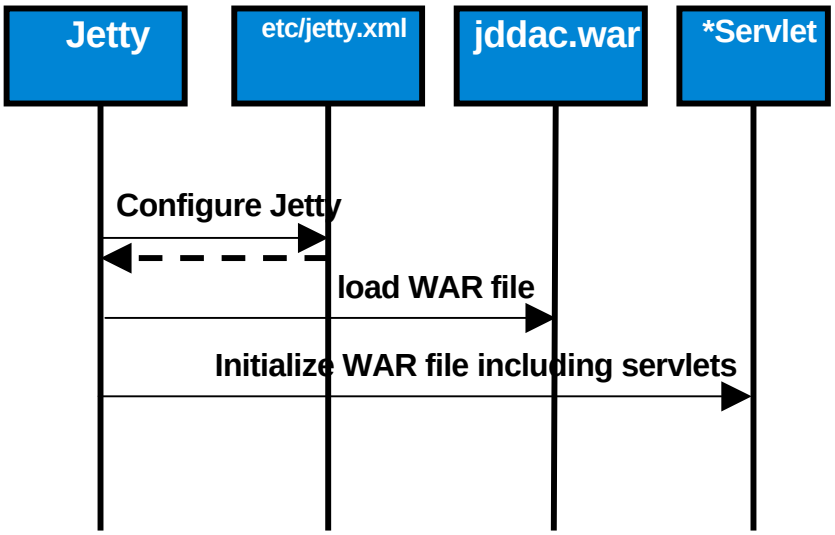
2. GetMeasurementMetadata - Returns list of metadata for specific measurement.
   - object -
   - method -
   - p0 - ID of measurement
   -
3. GetNRSS - Returns NRSS URI for a specific measurement.
   - object -
   - method -
   - p0 - ID of measurement
4. DeleteMeasurement - Deletes a specific measurement.
   - object -
   - method -
   - p0 - ID of measurement

24 Probe Startup Sequence

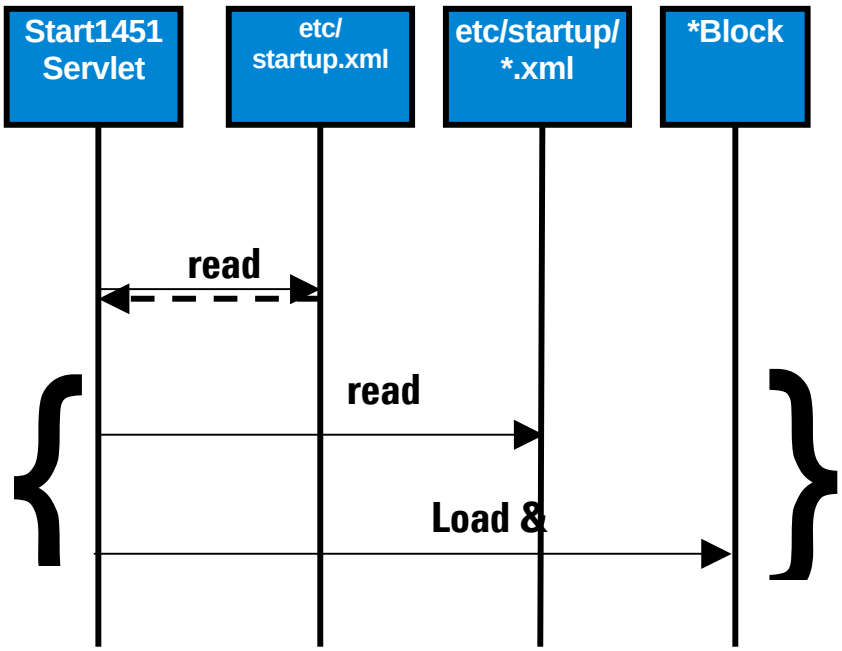
25 Probe Measurement Flow

26 Server Startup

This chapter describes the sequence of events that happen during the JDDAC server startup.



This is all pretty much standard J2EE servlet container startup sequence. Next, we start the JDDAC specific start up sequences.



The 1451Servlet is used to bootstrap the initialization process. It reads the XML configuration file located in etc/startup.xml. This file may include other configuration XML files which are typically located in the etc/startup directory. These XML files direct the JDDAC server to create F-Blocks and initialize their attribute values.

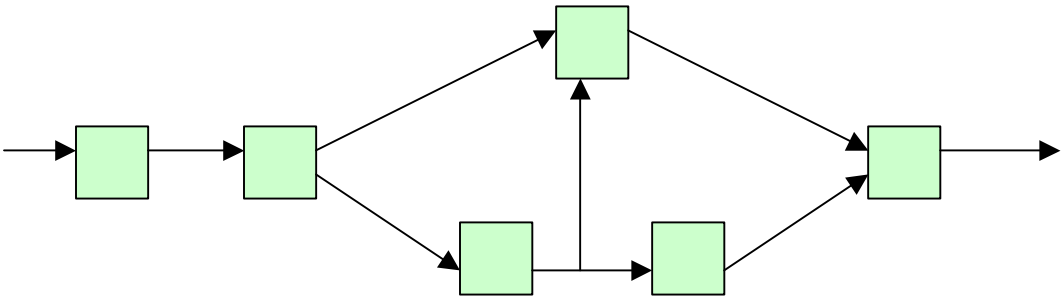
27 Server Measurement Flow

28 Adding an F-Block

28.1 Introduction

This note note will show you how to create and add an F-Block to a JDDAC application.

An F-Block, short for Function Block, contains measurement processing functionality. It accepts measurement data as input, performs processing on them, and outputs the processed measurement data. A JDDAC application contains multiple instances of different kinds of F-Blocks wired together to form a measurement processing chain. Within this chain, the measurement data flows from block to block, being transformed or processed as they are inside an F-Block. F-Blocks can be seen as an atom which makes up a JDDAC application.



In this tutorial, we will show you how to write an F-Block and how to get it linked into a JDDAC dataflow application. In particular, we will create a simple F-Block which accepts data, prints what it receives on the screen, modifies the data, and then outputs the modified data for the next block.

There are two communication models that connect the F-Blocks: publish-subscribe and client-server. In this tutorial, we will show how to use the publish-subscribe communication model.

F-Blocks inherit from the class net.java.jddac.jmdi.block.FunctionBlock. There are four key methods of interest in this class: perform(), configure(), notify(), and publish(). The perform() method enables it to receive client-server notifications which we will not be using in this tutorial. The configure() method is called by the start-up configuration subsystem. The other two methods deal with the publish-subscribe communication model.

When a F-block receives a publication, its notify() method is called. And when it is ready to emit data, it calls its publish() method.

28.2 A New F-Block

To build a new F-Block, you write a new class inherited from net.java.jddac.jmdi.block.FunctionBlock., then override a few of the methods as shown below.

Let's build a simple class with the three relevant methods. We won't worry about the perform() method in this example because we don't care about client-server here. We'll let the implementation in the base class worry about it.

```
import net.java.jddac.jmdi.block.*;

public class SimpleBlock extends FunctionBlock
{
      public SimpleBlock ();
      public boolean configure(ArgArray configuration)
                                          throws Exception;
            public void notify(short publication_id,
                               String publicationTopic,
                         ArgArray payload);
      protected void publish(int portNum, ArgArray
publication_contents)
                              throws OpException;

}
```

Now let's fill in the internals:

The configuration subsystem allows the application deployer to pass runtime parameters into the blocks. This is how we get at the parameters. I will show how to write the configuration file a little later.

```
String datum;
datum = configuration.getString("MyData");
```

You can use the mechanism to get at numbers as well as text.

What we want is to receive a publication, append data to it, and pass it on. This may be some sort of a timestamp block that adds its own attribute to every measurement publication it receives and then passes it on to blocks further down the pipeline.

So let's start with the notify() method since that is where the publications come into the system.

```
payload.add("Simple", datum);
publish(1,payload);
```

The fundamental data structure in JDDAC is an ArgArray. An ArgArray is a collection of name/value pairs akin to a hashtable in Java, where the name is of type String and value is any Java object. The parameter *payload* is the incoming ArgArray that the F-Block receives. And what we do is add another name/value pair with the name of "simple" to the payload, and then publish the modified ArgArray on output port 1 of the Block.

These few lines are sufficient to build a new F-Block that can be added to a JDDAC application! Obviously, more complex algorithms will require more complex F-Blocks. Putting all the code fragments together, the simple F-block Java source file looks like this:

```
import net.java.jddac.common.type.*;
import net.java.jddac.jmdi.block.*;

public class SimpleBlock extends FunctionBlock
{
      public String datum;

      public SimpleBlock ()
      {

      }

      public boolean configure(ArgArray configuration)
                                    throws Exception
      {
            // get the data we want to append from the runtime
configuration system
            datum = (String) configuration.get("MyData");
      }

            public void notify(short publication_id,
                        String publicationTopic,
                  ArgArray payload)
      {
            // Create copy of payload since it is read only
            ArgArray mycopy;

            Payload.cloneContentsTo(mycopy);

            // add our own Simple data to the incoming measurement
data
            mycopy.add(“Simple”, datum);

            // publish it on port 1 to other F-Blocks.
            publish(1, mycopy);
      }
}
```

Of course, the question still remains, what is the ‘1’ in the call to publish(), and how does this simple component get connected to the rest of the F-Blocks in the system. This is where the configuration comes in.

28.3 Configuring F-Blocks

The configuration for all F-Blocks are kept in a file called startup.xml. Inside startup.xml, there is a fragment for each F-Block. Here is one for the SimpleBlock F-Block.

```
<argArray>
```

```
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">simple</arg>
  <arg n="Entity.objectName">My Simple Block</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >1</arg>
  <arg n="Block.s1.qual">incoming</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">outgoing</arg>
  <arg n="MyData">Cow</arg>
<argArray>
```

Here is an explanation of the required descriptors for an F-Block.

- **className** – this is the name of the class
- **instanceName** – this is the name of the object. This is how other objects in the system refer to this object, if there is need to do so.
- **Entity.objectName** – this is a description name of the object. You can put anything you like here.
- **Entity.owningBlockObjectTag** – This denotes block ownership. Most of the time, it is best to put 'NCAP' here.
- **Block.numSubs** – This denotes how many incoming data ports this block has.
- **Block.s1.qual** – This is the name of the subscription topic for the first incoming data port. If there were 3, they would be called Block.s1.qual, Block.s2.qual, and Block.s3.qual.
- **Block.numPubs** – Conversely, this denotes how many outgoing data ports that this block has.
- **Block.p1.topic**, like the subscriptions, this is the name of the publication topic for the outgoing data port. If there were multutiple output ports, they would be referred to as Block.p1.topic, Block.p2.topic, and Block.p3.topic.
- **MyData** – this is a parameter specific to this block.

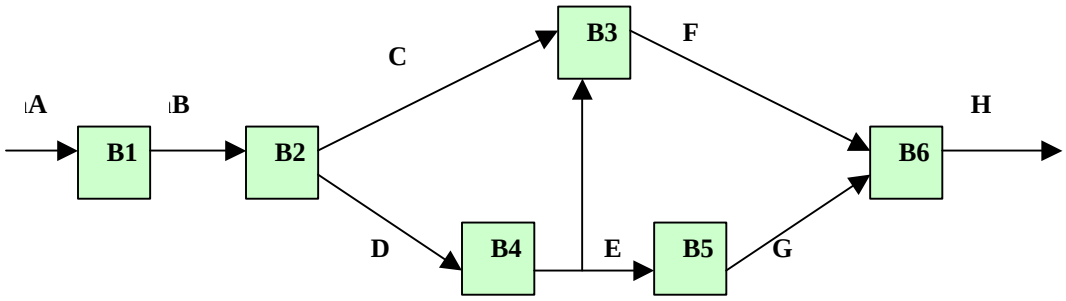
For our simple block, we have one incoming publication on the topic called "incoming", and we have an outgoing publication called "outgoing". The

There is a fragment of this XML for each instance of the object, not each type. If we have two such fragments, the application will have two such blocks. And by setting the names of the blocks appropriately, we can link them into a processing chain.

For more details on the configuration system, click here.

28.4 Multiple F-Blocks Example

Let's link several of these blocks together into a processing chain like the one shown at the top of the document.



Here is what the XML to connect these may look like:

```
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B1</arg>
  <arg n="Entity.objectName">Block #1</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >1</arg>
  <arg n="Block.s1.qual">A</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">B</arg>
  <arg n="MyData">Cow</arg>
<argArray>
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B2</arg>
  <arg n="Entity.objectName">Block #2</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >1</arg>
  <arg n="Block.s1.qual">B</arg>
  <arg n="Block.numPubs" t="int" >2</arg>
  <arg n="Block.p1.topic">C</arg>
  <arg n="Block.p2.topic">D</arg>
  <arg n="MyData">Dog</arg>
<argArray>
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B3</arg>
  <arg n="Entity.objectName">Block #3</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >2</arg>
  <arg n="Block.s1.qual">C</arg>
  <arg n="Block.s1.qual">E</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">F</arg>
  <arg n="MyData">Cat</arg>
<argArray>
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B4</arg>
  <arg n="Entity.objectName">Block #4</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >1</arg>
  <arg n="Block.s1.qual">D</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">E</arg>
  <arg n="MyData">Cow</arg>
<argArray>
```

```
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B5</arg>
  <arg n="Entity.objectName">Block #5</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >1</arg>
  <arg n="Block.s1.qual">E</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">G</arg>
  <arg n="MyData">Cow</arg>
<argArray>
<argArray>
  <arg n="className">SimpleBlock</arg>
  <arg n="instanceName">B6</arg>
  <arg n="Entity.objectName">Block #6</arg>
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="Block.numSubs" t="int" >2</arg>
  <arg n="Block.s1.qual">F</arg>
  <arg n="Block.s1.qual">G</arg>
  <arg n="Block.numPubs" t="int" >1</arg>
  <arg n="Block.p1.topic">H</arg>
  <arg n="MyData">Cat</arg>
<argArray>
```

Note that some of these blocks may have two subscriptions and one publication, while others have one publication and two subscriptions.

29 Adding a Transducer

29.1 Introduction

In this tutorial, we will show you how to create an object implementing the TIMInterface (Transducer Interface Module) and how to get it linked into a JDDAC dataflow application. In particular, we will create a simple object which measures two aspects of the currently running JVM.

In the latter half of the tutorial, we will discuss the different methods available on the TIMInterface used to retrieve measurements from an object implementing the TIMInterface interface.

29.2 Overview

A transducer implements the TIMInterface in order to provide a connection between some sort of real world sensor or actuator and the JMDI Measurement Dataflow API. The JDDAC libraries provide an abstract helper class called BasicTIM which implements much of the nitty gritty details and simplifies the interface to just a handful of methods.

Each transducer is responsible for providing a set of meta data or TEDS (Transducer Electronic Data Sheet) to the infrastructure to allow the measurements to be self describing. This meta data is a key element used in much of JMDI.

In order to simplify the notion of values we call each separate sensor or actuator value a “channel” and treat it as a separate entity. A transducer thus will have a set of channels and we further associate a numeric index to each channel and following the IEEE1451.2 numbering scheme we assign the first channel as one and increment from there. Thus, a three channel transducer will have channel numbers 1, 2, and 3. In the discussion which follows the notion of a channel should be interchangeable with the notion of a single sensor value or actuator value in the transducer.

29.3 A New Transducer

To build a new transducer, write a new class inherited from net.java.jddac.jti.BasicTIM., then override a few of the methods as shown below. We won’t worry about actuators in this and will focus instead only on sensors. We’ll let the implementation in the base class worry about most of the details.

```
import net.java.jddac.*;
public class MemoryExample extends BasicTIM {

   public MemoryExample();
   protected Measurement readChannel(int chanNum);
}
```

The `MemoryExample` method is the constructor while the `readChannels` method does the real work. Now let’s fill in the internals. In order to make the measurement

results from the transducer 'self describing' some information needs to be included with each measurement channel such as name, units, data type etc. This is done by utilizing the `setChannelTeds` method in the `BasicTIM` class.

The `setChannelTeds` method takes as an argument an array settings for each value in the transducer. Each of these in turn is itself an array that contains name/value pairs of associated meta data. For example, to describe a channel which represents the amount of free memory in the JVM we would create an set of name/value pairs as follows:

```
Object channelInfo[] = {
    MeasAttr.NAME,"freeMemory",
    MeasAttr.DESCRIPTION,"JVM Free Memory",
    MeasAttr.UNITS,"Bytes",
    MeasAttr.DATA_TYPE, TypeAttr.INTEGER64 }
```

Here we use some pre-defined tags found in the MeasAttr class for NAME, DESCRIPTION, UNITS, and DATA_TYPE. Since our transducer will have two channels we create an array to hold the meta data for all our channels:

```
private final Object teds[][] = {
        // channel 1
        {
                MeasAttr.NAME,"freeMemory",
                MeasAttr.DESCRIPTION,"JVM Free Memory",
                MeasAttr.UNITS,"Bytes",
                MeasAttr.DATA_TYPE,"Integer64" },
        // channel 2
        {
                MeasAttr.NAME,"totalMemory",
                MeasAttr.DESCRIPTION,"JVM Total Memory",
                MeasAttr.UNITS,"Bytes",
                MeasAttr.DATA_TYPE,TypeAttr.INTEGER64 })
```

Now that we have our meta data constructed we will then add a registration invocation to our constructor:

```
public MemoryExample(){
        setChannelTeds(teds);
}
```

Now that we have provided the `BasicTIM` with a notion of our meta data we can work on producing some actual measurement results in the `readChannel` method. The `readChannel` method is provided with an integer argument which specifies the channel to be read. Since the return type from `readChannel` is a Measurement, all we have to do is create a new `Measurement` and place a value into the newly constructed `Measurement` instance:

```
            Measurement newMeas = new
   Measurement(lastChanValues[chanNum]);
            long result=0;
            switch (chanNum) {
```

```
                case 1 : result=Runtime.getRuntime().freeMemory();
                break;
                case 2: result=Runtime.getRuntime().totalMemory();
                break;
        }
        newMeas.put(MeasAttr.VALUE,new Long(result));
        return newMeas;
```

Let's look at this line by line. Each `Measurement` result is expected to contain all pertinent meta data for the measurement so the new `Measurement` instance is created initially as a copy of the last value produced which had full meta data:

```
new Measurement(lastChanValues[chanNum]);
```

When a new measurement is constructed with an existing measurement as an argument, a clone is done of the `Measurement` container so all prior meta data placed into the `Measurement` object is retained. This meta data is initially loaded into the `lastChanValues` instances when the `setChannelTeds` method is used.

The code then makes a decision on which channel is being requested and generates a long result based on the channel number. Finally, the new result is placed into the `newMeas` object as a name/value pair and then returned to the caller.

```
newMeas.put(MeasAttr.VALUE,new Long(result));
return newMeas;
```

Every measurement in JMDI is expected (but not required) to have a time stamp associated with the measurement as additional piece of meta data so let's add a timestamp to our measurement:

```
TimeRepresentation timestamp = new TimeRepresentation();
newMeas.put(MeasAttr.TIMESTAMP, timestamp);
```

Our final method and entire class then looks as follows:

```
import net.java.jddac.*;
public class MemoryExample extends BasicTIM {
   public MemoryExample(){
         setChannelTeds(teds);
   }
   protected Measurement readChannel(int chanNum) {
         Measurement newMeas = new
Measurement(lastChanValues[chanNum]);
         TimeRepresentation timestamp = new
TimeRepresentation();
         newMeas.put(MeasAttr.TIMESTAMP, timestamp);
         long result=0;
         switch (chanNum) {
               case 1 :
result=Runtime.getRuntime().freeMemory();
               break;
               case 2:
result=Runtime.getRuntime().totalMemory();
               break;
         }
```

```
                    newMeas.put(MeasAttr.VALUE,new Long(result));
                    return newMeas;
                }
            }
```

29.4 Configuring Transducers

The configuration for a specific transducer can be kept in an xml file that adheres to the JDDAC configuration api specification. Here is one for our example transducer:

```
<argArray>
  <arg n="className">MemoryExample</arg>
  <arg n="instanceName">MyMemoryExample</arg>
  <arg n="TIM.id">JVM-0908001234</arg>
<argArray>
```

Here is an explanation of the required descriptors for an F-Block.

- **className** – this is the name of the class
- **instanceName** – this is the name of the object. This is how other objects in the system refer to this object, if there is need to do so.
- **TIM.id**– this is a unique identifier for measurements produced by this instance. It should be unique within the context of a JDDAC NCAP. In most situations this property does not need to be set as the BasicTransducerBlock will assign this value.
- When a transducer is used in conjunction with a TransducerBlock such as the BasicTransducerBlock, the BasicTransducerBlock xml will specify an association with a particular transducer:

```
<argArray l="9">
  <arg n="Entity.owningBlockObjectTag">NCAP</arg>
  <arg n="BasicTransducerBlock.transducerName">MyMemoryExample</arg>
  <arg n="Block.p1.topic">storeMeasurement</arg>
  <arg t="int" n="BasicTransducerBlock.period">10000</arg>
  <arg
n="className">net.java.jddac.jmdi.block.BasicTransducerBlock</arg>
  <arg t="boolean" n="BasicTransducerBlock.autoStart">true</arg>
  <arg n="instanceName">My Memory Usage TBlock</arg>
  <arg n="Entity.objectName">Demo Basic Transducer Block for Memory
Usage</arg>
  <arg t="int" n="Block.numPubs">1</arg>
</argArray>
```

For more details on the configuration system, click here.

29.5 Using Transducers

The TIMInterface objects typically used by TransducerBlocks. Typically, these are the actions you can do with a TIMInterface object:

- Read values from sensors

- Write values to actuators
- Query for metadata
- Register notification for events

To read a value, first invoke the startRead() method. If the transducer supports asynchronous reads it will queue up the read and subsequently notify objects registered with the addDataReadyCallback function. Transducers that do not inherently support an asynchronous read will spawn a separate thread to do a blocking read and notify DataReadyInterface listeners.

Next, invoke either a readLast() to retrieve a queued measurement, or a read() to cause a measurement to occur. The method readLast() simply returns a previously measured value and thus does not block. A call to read(), however, may block if the underlying call to the actual physical hardware is blocking in nature.

Writes work in a similar fashion, except with the methods startWrite() and write().

Metadata is retrieved via a getMetadata() method. The ID for the metadata is retrieved via getMetaID().

Lastly, clients of the TIM that wish to be notified when data is available should implement the DataReadyInterface interface and register themselves for notification with the TIM via the addDataReadyCallback() method.

30 How to Add Web Pages to Server

31 Introduction to Measurement Calculus

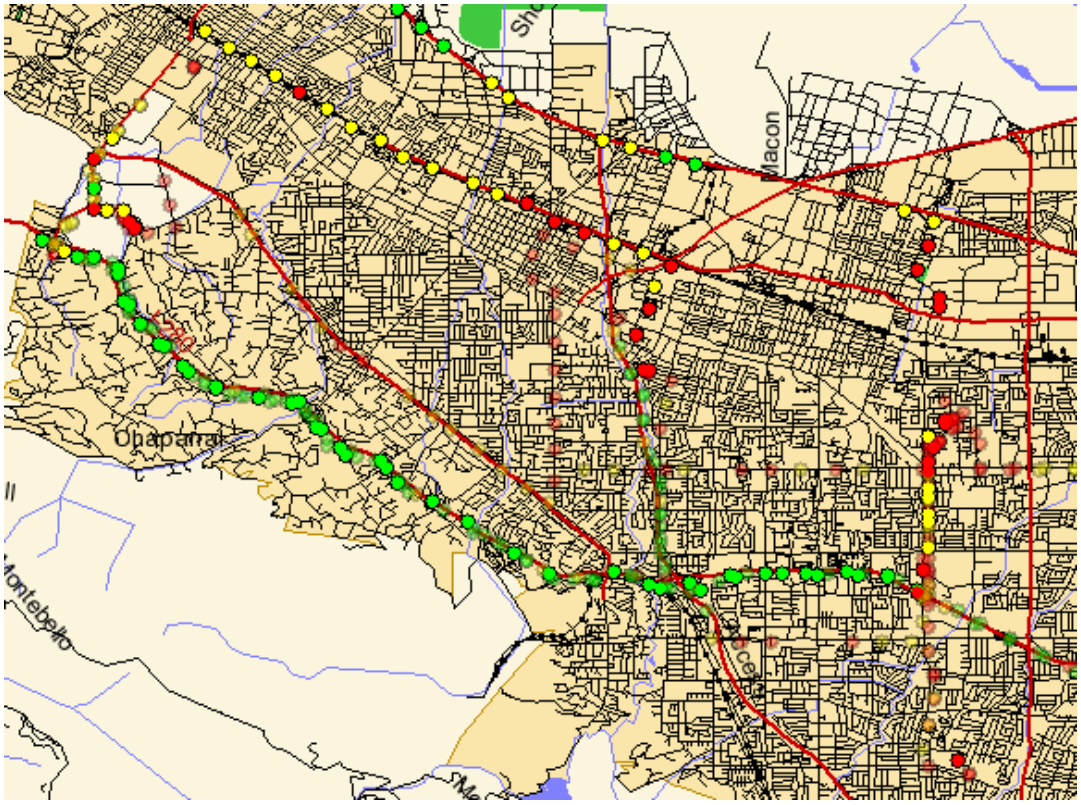
32 Sample Applications

This section describes two sample applications which have been built on the JDDAC platform.

32.1 Traffic Monitoring

In this application, JDDAC probes are deployed onto J2ME enabled cellular phones with GPS positioning capability. The handsets are carried around in cars, and the JDDAC probes make periodic measurements of travel velocity and direction of the handset as it travels with the car. The handset's velocity becomes an indication of how well the traffic is flowing. These measurements are reported, via the cellular network, to a J2EE-based JDDAC server, persisted, and plotted on a map.

With a large enough population of phones all making these traffic measurements, we can obtain an almost instantaneous picture of the traffic conditions across any particular region.

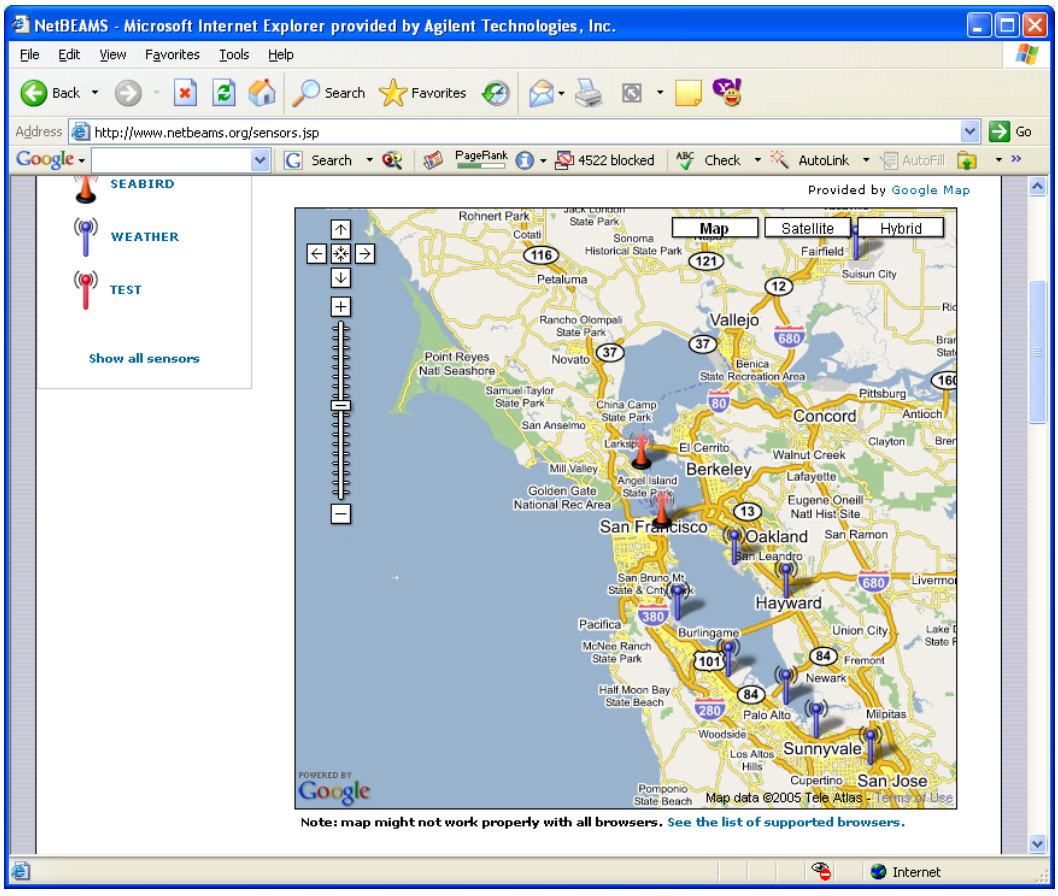


The map above is a screenshot of the traffic in the Silicon Valley area, with red indicating speeds of less than 35 mph, green above 60 mph, and yellow in between.

32.2 NetBEAMS

NetBEAMS (Networked Bay Environmental Assessment Monitoring Stations) is collaboration between Romburg Tiburon Center, San Francisco State University, Agilent Technologies, and Sun Microsystems. It is a project to deploy a network oceanographic instruments under the waters of San Francisco Bay to collect water quality data such as temperature, pressure, salinity as an effort to help oceanographers better understand the bay.

The system is composed of a JDDAC probe running on a cell phone connected via a serial cable to the marine instrument located under water. The JDDAC probe on the phone queries the instrument for data and stores it in the internal memory of the phone. At periodic intervals, the phone transmits the data to the JDDAC server, where the data is stored in a MySQL database for the scientists. The data is also made available via a NRSS data feed to be plotted on a map on a web site.



33 XML Schema

This is the JDDAC XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
targetNamespace="http://jddac.labs.agilent.com/jddac1v0"
            xmlns="http://jddac.labs.agilent.com/jddac1v0"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsd:simpleType name="jddacArgTypes">
        <xsd:annotation>
            <xsd:documentation>
            This type defines the possible values for the t
attribute of the arg element
            </xsd:documentation>
        </xsd:annotation>

        <xsd:restriction base="xsd:string">
            <xsd:enumeration value=""/>
            <xsd:enumeration value="int"/>
            <xsd:enumeration value="long"/>
            <xsd:enumeration value="float"/>
            <xsd:enumeration value="boolean"/>
            <xsd:enumeration value="id"/>
            <xsd:enumeration value="units"/>
            <xsd:enumeration value="timeRepresentation"/>
            <xsd:enumeration value="intArray"/>
            <xsd:enumeration value="longArray"/>
            <xsd:enumeration value="booleanArray"/>
            <xsd:enumeration value="byteArray"/>
            <xsd:enumeration value="floatArray"/>
        </xsd:restriction>
    </xsd:simpleType>

   <xsd:simpleType name="jddacArgArrayTypes">
        <xsd:annotation>
            <xsd:documentation>
            This type defines the possible values for the t
attribute of the argArray element
            </xsd:documentation>
        </xsd:annotation>

        <xsd:restriction base="xsd:string">
            <xsd:enumeration value=""/>
            <xsd:enumeration value="vector"/>
            <xsd:enumeration value="record"/>
            <xsd:enumeration value="measurement"/>
            <xsd:enumeration value="stringArray"/>
            <xsd:enumeration value="location"/>
        </xsd:restriction>
```

```
    </xsd:simpleType>

   <xsd:attributeGroup name="jddacArgAttr">
        <xsd:annotation>
            <xsd:documentation>
            This defines the possible attributes for the arg
element
            </xsd:documentation>
        </xsd:annotation>

         <xsd:attribute name="n" type="xsd:string"/>
         <xsd:attribute name="t" type="jddacArgTypes"/>
         <xsd:attribute name="l" type="xsd:nonNegativeInteger"/>
   </xsd:attributeGroup>

   <xsd:attributeGroup name="jddacArgArrayAttr">
        <xsd:annotation>
            <xsd:documentation>
            This defines the possible attributes for the argArray
element
            </xsd:documentation>
        </xsd:annotation>

         <xsd:attribute name="n" type="xsd:string"/>
         <xsd:attribute name="t" type="jddacArgArrayTypes"/>
         <xsd:attribute name="l" type="xsd:nonNegativeInteger"/>
   </xsd:attributeGroup>

   <xsd:complexType name="jddacArg">
        <xsd:annotation>
            <xsd:documentation>
            This defines the xml type for the arg element
            </xsd:documentation>
        </xsd:annotation>

         <xsd:simpleContent>
               <xsd:extension base="xsd:string">
                     <xsd:attributeGroup ref="jddacArgAttr"/>
               </xsd:extension>
         </xsd:simpleContent>
   </xsd:complexType>

   <xsd:complexType name="jddacArgArray">
        <xsd:annotation>
            <xsd:documentation>
            This defines the xml type for the argArray element
            </xsd:documentation>
        </xsd:annotation>

         <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element name="argArray"
type="jddacArgArray"/>
               <xsd:element name="arg" type="jddacArg"/>
         </xsd:choice>
```

```
        <xsd:attributeGroup ref="jddacArgArrayAttr"/>
    </xsd:complexType>

    <xsd:element name="argArray" type="jddacArgArray">
        <xsd:annotation>
            <xsd:documentation>
            This is the outer element definition
            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
</xsd:schema>
```

34 List of Blocks

This chapter lists the F-Blocks that come in the JDDAC distribution that you can use to build your own applications.

Name	Path	Description
MessageQueueBlock	Jmdi/j2se/src/net.java.jddac.jmdi.fblock	Queues up messages for back channel
ServerMessageStore	Jmdi/j2se/src/net.java.jddac.jmdi.fblock	
TransportBridgeBlock	Jmdi/j2se/src/net.java.jddac.jmdi.fblock	Interface between the communication pipeline and the rest of the blocks on server
EchoBlock	Jmdi/common/src/net.java.jddac.jmdi.fblock	Echos back received publication. Used for 'ping' tests.
PubDiagBlock	Jmdi/common/src/net.java.jddac.jmdi.fblock	
Reporter	Jmdi/common/src/net.java.jddac.jmdi.fblock	Used on probe to communicate with server.
JFreeCharthBlock	Custom/j2se/src/net.java.jddac.jmdi.fblock	Generates charts with JFreeChart
SystemSetupBlock	Custom/j2se/src/net.java.jddac.jmdi.fblock	Sets system properties from configuration files.
AlarmBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Checks incoming publications for alarms.
DbConMgrBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	

GasPriceStoreBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Application-specific block to store gas prices into MySQL.
GenericStoreBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Generic storage block to persist ArgArray into MySQL.
JFreeChartBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Uses ArgArray to generate charts with JfreeChart.
NRSSBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Generates NRSS Feed
LogJdbcHandlerBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	
MeasSummaryBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Emits measurement attributes in the form of ArgArray
MeasTableStoreBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Stores measurements in MySQL.
ProbeBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Track probe checkin and stores probe attributes.
ReprtDbAccess	Custom/j2ee/src/net.java.jddac.jmdi.fblock	
ReportStoreBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Stores and provides measurement report definitions.
UserBlock	Custom/j2ee/src/net.java.jddac.jmdi.fblock	Stores user information

35 Detailed Block Descriptions

This section provides detailed descriptions of specific Blocks in the distribution.

35.1 ConditionEval

The ConditionEval class evaluates user specified conditions and generates either a boolean or object (e.g a String) result. The use of the result is dependent on the context of the ConditionEval. In the case of Jddac Initialization, ConditionEval boolean values are used to control loading and initialization of classes based on the existence of other object or property values. Alternatively, the ConditonEval may be used to pull values out of the environment for use as Strings or other configuration values. The following example will conditionally create an instance of com.example.MyClass if the microedition.profiles property exists.

```
<argArray>
    <arg n="className">com.example.MyClass</arg>
    <arg n="instanceName">George</arg>
    <argArray n="condition" t="vector">
        <argArray>
            <arg n="type">property</arg>
            <arg n="name">microedition.profiles</arg>
            <arg n="exists">true</arg>
        </argArray>
    </argArray>
</argArray>
```

When the condition specification is a vector the result is assumed to be a boolean created by the logical AND of each element in the vector. Since the above condition vector is a single value, it could also have been represented as a named ArgArray element:

```
<argArray n="condition">
        <arg n="type">property</arg>
        <arg n="name">microedition.profiles</arg>
        <arg n="exists">true</arg>
</argArray>
```

Thus, each condition specification must either be a vector of conditions evaluated to boolean using an AND operation or it must be an ArgArray with a single condition element. When an ArgArray is used to express a condition a variety of elements are supported:

- if, and, or, not and equals boolean elements
- invert, exists, and default result modifiers
- object element to retrieve an object from the registry
- property element to retrieve a system or NCAP property value
- containedBy element to test containment (location, numeric, string)
- concat element for string concatenation
- argExtract for tree based retrieval of an object from another ArgArray

Each condition element must provide an element named "type" to specify the condition as well as additional attributes that further specify how the element should be evaluated. The 'default', 'invert', and 'exists' modifiers can be applied to any of the condition elements and have the following meanings:

Attribute	Meaning
default	Specifies the default value of condition.

invert	Inverts the logical meaning of a condition.
exists	Tests for a non-null object reference. This modifier is useful with the argExtract, object, property, concat, and ifobj conditions.

35.1.1 Condition Types

The table below illustrates the condition types supported. Each condition requires one or more named attributes to fully specify the behaviour.

Condition type	attribute name	attribute type	Description
and			The `and` element has a single attribute that specifies a vector of conditions to be evaluated. This condition is true if all of its contained conditions are, conditions will be evaluated in the order they have been specified in the vector and processing stops on the first false condition.
	cond	vector	A list of conditions to be evaluated.
argExtract			Extract an object from an ArgArray. The `fields` attribute specifies how to navigate the `ArgArray` to extract the value.
	fields	vector	list of names to recursively decend into the ArgArray and extract elements. If a Vector is encountered in the ArgArray then the field name is treated as an ordinal index into the Vector.
concat			Concatenate as strings a list of items in the contained vector. Any items that are null are ignored.
	list	vector	A list of items that are converted to string and concatenated.
containedBy			Determines if a value is contained by the limites specified by `lowerBound` and `upperLimit`. If the object is a `Location` instance then the lowerBound and upperBound are presumed to be two corners of a square. If the object is a numeric type then `lowerBound` and `upperLimit` specify the lower and upper bounds which must contain the object. That is, `lowerBound` <= object <=

			`upperLimit`. If either the `lowerBound` or the `upperBound` is not specified then it is not utilized in the comparison. This allows a less than test or a greater than test by only specifying one of the attributes.
	value	varies	A value to check against the specified boundary.
	lowerBound	varies	The first boundary.
	upperBound	varies	The second boundary.
	list	varies	A list of allowable values. The list can be any suitable array type among `byte[]`, `int[]`, `long[]`, `Location[]`, `Float64[]`, `String[]`, or `Object[]`. If the value does not match any items in the list then the condition is considered to be false. If the list is a `String` array then a simplified regex pattern match is used where the only regex pattern supported is '.*' (wildcard).
equals			Determines if two values are equal. The `equals` method on the object is used for the determination unless the value is a `String`. If the value is a `String` then a simplified regex pattern match is used where the only regex pattern supported is '.*'.
	v1	varies	The first value to compare against.
	v2	varies	The second value to compare against.
if			Evaluates an expression to a boolean result and then chooses either a then clause or an else clause as the result. If the expression is true and the then clause is not specified, the result is considered to be the result of the if expression.
	expr	varies	An expression that is evaluated to a boolean.
	then	varies	The result when `expr` evaluates to true.
	else	varies	The result when `expr` evaluates to false.
ifobj			Evaluates an expression and considers it to be true if it is not null. If the expression is true and the then clause is not specified, the result is considered to be the result of the if expression

			before conversion to boolean. This is most often used in conjunction with argExtract to check for the presence of a field. It can also be used to check the existance of properties.
	expr	varies	An expression considered true if it is not null.
	then	varies	The result when `expr` evaluates to true.
	else	varies	The result when `expr` evaluates to false.
not			Invert the specified condition
	cond	ArgArray	A single condition to be evaluated.
or			The `or` element has a single attribute that specifies a vector of conditions to be evaluated. This condition is true if any of its contained conditions are, conditions will be evaluated in the order they have been specified in the vector and processing stops on the first true condition.
	cond	vector	A list of conditions to be evaluated.
object			
	name	String	The name of an object to look up in the object registry.
property			Determine the existance of a system property using the System#getProperty method. If the System#getProperty method returns null then the NCAP registry is queried for an object of the same name. If the property does not exist then false is returned. If `value` is specified then the property must exist and contain the string specified by `value` to return true.
	name	String	The name of property to query.
	value	varies	A string which must match the property using a regex compare that only supports '.*' wildcards.

35.2 Reporter
35.3 NCAP
35.4 BasicTransducer
35.5 BasicTIM
35.6 GenericStore
35.7 GenericArgArrayStore
35.8 MeasTableStore
35.9 MetadataStore
35.10 ProbeCheckin
35.11 ReportDBAccess

36 FAQ

This note provides a FAQ (Frequently Answered Questions) for JDDAC.

General

3) What is JDDAC?

JDDAC stands for Java Distributed Data And Acquisition. To find out more about it, visit http://jddac.dev.java.net

Installation and Building

4) How do I get the JDDAC source code?

Visit http://jddac.labs.agilent.com/jddac/downloads and download the ZIP file containing the source code.

5) What are the requirements to run JDDAC?

To run JDDAC on the J2SE, you will need JDK 1.5 or later. To run it on J2ME, you will need a platform which provides MIDP 1.0 and CLDC 1.0 or later.

6) How do I build the JDDAC source code?

You can either use ant or Eclipse.

Platforms

7) Does JDDAC work with J2ME?

Yes.

8) What compromises have been made for JDDAC to work with J2ME?

There is a separate class hierarchy to deal with floating point numbers. Thus, we do not use the classes such as Float available in java.lang because these classes are not available in J2ME MIDP 1.0.

Architecture

9) What is a TIM?

Transducer Interface Module. It is the class used to interface to transducers. This is a term from the IEEE standard 1451.0.

10) What is a F-Block?

It's a nickname for Function Block. It is the class used to contain measurement processing capabilities. This is a term from the IEEE standard 1451.1.

11) What is TEDS?

Transducer Electronic data Sheet. It is a machine readable description of a transducer which describes the attributes of the transducer, how to communicate with it, and the metadata of the measurements it generates or the actuator settings that it accepts.

12) What is the default XML format used by JDDAC?

The XML schema used is based on the data format from IEEE 1451.1.

13) Why does the XML format use only two tag names?

Because we wanted the set of entities represented in the XML to be extensible without needing to update the schema. By specifying the different entities with different attributes of the same tag, rather than updating the tag name, we avoid a tag namespace explosion in our schema.

14) What is a JDDAC ID?

A JDDAC ID is a URI used to identify objects and measurements within a JDDAC application.

Developing Components

15) How do I create a new TIM?

Inherit from the TIM class. You will need to fill out the TEDS array to specify the TEDS information for each channel, and then implement the readChannel() method to provide the data for each channel.

16) How do I create a new F-Block?

Inherit from the FunctionBlock class. If you wish to receive publications, you will need to implement the notifySubscriber() method. If you wish to receive client-server invocations, you will need to implement the perform() method.

17) How do I connect F-Block inputs and outputs?

The configuration is set in the configuration file.

18) What are the possible fields I can put inside a Measurement?

Look in MeasAttr class.

19) How do I send a measurement to the server?

Configure the Reporter and subscribe to a particular topic, and then publish on that topic.

20) How do I receive commands from the server?

Subscribe to topics from networks.

21) How do I publish on a topic?

From within an F-Block, call the publish() helper method.

22) How do I receive a publication?

There are two things you need to do. You need to properly configure the block in the configuration XML to receive the publication on the appropriate port. You also need to implement the notifySubscriber() method in the SubscriberCallback interface to do something with the publication.

23) what's the difference between TIM and F-Block?

A TIM is used to interface to transducers and contains functionality to interact with TEDS and software drivers for transducers. An F-Block is a container for data processing functionality. They can both be wired to send and receive publications, however.

Assembling and Configuring Applications

24) What is the naming scheme for jar files created in the lib directory?

The names go something like <interface name>-<platform name>.jar. There are some classes which are core to all JDDAC interfaces and thus are assigned an interface name of 'core'. The classes which are common to all JDDAC platforms are assigned a platform name of 'common'.

These are the different possible interface names:

- common
- probe
- portal

These are the possible platform names.

- common
- j2me
- j2se
- portal

Lastly, the junit tests for the JAR files have the string '-junit' appended to the name corresponding to the JAR that they contain tests for.

Deployment

25) How do I start my own new application?

Run the jar file. On the J2SE platform, invoke the JVM with a command similar to

```
java -jar JddacClient.jar
```

On the J2ME platform, your mileage will vary depending on which phone emulator you use.

Debugging

26) Why am I not receiving any subscription?

Check the configuration in the configuration XML file. Most of the time, there is a mismatch between the names used in the publication versus the subscriptions.

Miscellaneous

27) Does JDDAC use JXTA?

No, although there exists JDDAC applications that make use of JXTA.