

ORB Notes

Ken Cavanaugh

Version 0.4

Abstract

Comprehensive documentation on the Sun Java ORB.

Contents

1	Starting Points	1
1.1	Changes for the GlassFish-CORBA project	1
1.2	ORB Development procedures	1
1.3	Workspace Structure and Builds	2
1.3.1	docs	2
1.3.2	ORB source	3
1.3.3	ORB Renaming	5
1.3.4	ORB Build Files	5
1.3.4.1	Building the ORB	5
1.3.4.2	Structure of the build files	6
1.3.5	tests	6
1.3.6	libraries	7
1.4	ORB Developer Tests	7
1.4.1	Test Suites	7
1.4.2	debugging tests	10
1.5	ORB SQE Tests	10
1.6	Thoughts on Middleware goals	10
1.7	ORB Coding Practices	12
1.8	Supporting JDK and App Server	15
2	The ORB Class	16
2.1	Inheritance Structure	16
2.2	ORB SPI structure	16
2.3	ORB Initialization	18
2.3.1	The Configuration Framework	18
2.3.1.1	DataCollector	19
2.3.1.2	Operation	21
2.3.1.3	PropertyParser	22
2.3.1.4	Base Classes for Parsing Properties	23
2.3.2	Details of ORB.init	25
2.3.2.1	The ORB configurator	25
2.3.3	Initializing the ORB in the App Server	27
2.4	ORB Shutdown	27
3	Dispatch Path Overview	28
4	Presentation	29
4.1	Stubs and Skeletons	29
4.2	Data types	29
5	Encoding	30
5.1	Repository IDs	30

6	Protocol	31
6.1	Subcontract IDs	31
6.2	IORs	31
6.3	Service Contexts	31
6.4	GIOP Message Representation	31
7	Transport	32
8	Other Aspects of the ORB	33
8.1	Object Adapters	33
8.2	The RequestDispatcherRegistry	33
8.3	Encoding Details	33
8.4	ORB Logging	33
8.5	ORB Monitoring	33
8.6	ORB versioning	33
8.7	ORBD and Server Activation	33
	8.7.1 current model	33
	8.7.2 ideas for using ORT	33
8.8	Portable Interceptors	33
8.9	RMI-IIOP Implementation	33
8.10	Resolvers	33
8.11	Name Services	33
8.12	ORB and App Server Integration	33
9	Utilities	34
9.1	Fast Object Copying	34
9.2	Dynamic Code Generation	34
9.3	Useful utilities	34
9.4	FSM Framework	34
9.5	Graph Utilities	34
9.6	JDK 5 Specific Utilities	34
9.7	Timing Framework	34
10	Living with our legacy	35
10.1	Testing Principles	35
10.2	Benchmarking	35
10.3	FOLB Support	35
10.4	HWLB Support	35
11	Compilers	36
11.1	New rmic iiop backend	36
11.2	idlj	36
12	Future Directions	37
12.1	Embedded Languages	37
12.2	Components	37
12.3	Fast Marshalling	37
12.4	Security	37
12.5	Better handling of Invocation Info	37

Chapter 1

Starting Points

1.1 Changes for the GlassFish-CORBA project

Much of this document reflects policies and procedures that have been in place for some time, and so do not completely reflect the changes brought about by putting the CORBA source code out on java.net. I am updating this, but discussions about TeamWare and Sun-internal file systems clearly do not reflect the differences found in developing using CVS on java.net.

Note that all CORBA development has been done in the past using TeamWare. GlassFish-CORBA must use CVS, since that's mostly what's available on java.net. The TeamWare and CVS repositories will be synchronized automatically using a CVS-TeamWare bidirectional bridge, but that is not yet fully operational (as of this writing).

1.2 ORB Development procedures

Over the years, we have established a fairly successful development methodology for working on the ORB. Any piece of work from a minor bugfix to a major new subsystem iterates over the same stages one or more times. I'll discuss this briefly here, but the details will appear below.

1. Discuss or document the design. This can take many forms, ranging from an informal discussion to a written document. Written documents have in the past been done in many different formats, including email, text files, FrameMaker documents, StarOffice documents, HTML, XML and DocBook, and L^AT_EX (about the only thing none of us have used seems to be Word!). Going forward, all of these are still viable in different ways, but I'd really like to make sure that we capture all of the design efforts into our documentation collection. This is currently either in <ws>/docs in the CORBA workspace or at the CORBA web site at /java/j2ee/CORBA. In the future, everything should be in the workspace, with the web site linking to the workspace. I plan to make this document a major component of the documentation process (probably with the help of latex2html and similar tools).
2. Develop the code. This include writing both the code to implement the change and the code to test it. Bugs need a test that fails before the change is made, and that passes after the change. All tests must be incorporated somewhere in the CORBA test suites (see 1.4.1 on page 7) so that they can be run automatically. Test execution speed is important, so make the test as fast as possible. Please follow the principles discussed in 1.6 on page 10 and in 1.7 on page 12.
3. Verify that nothing has been broken by the change. This requires running all of the CORBA test suites, and fixing any problems that show up (see 1.4.2 on page 10 for help on debugging tests).
4. Prepare a webrev for a code review. The webrev tool is available on SWAN in /java/devtools/share/bin. Read the comments in this kshell script for details. (TBD: what tools should we use for CVS?)

5. Have at least 1 (and preferably more) developer from the GlassFish-CORBA team review the changes. Make any necessary corrections from the review, and re-review if necessary.

TBD: Currently we conduct all code reviews in real time, usually via phone conference (since this has been a distributed team for a long time). We need to investigate this in the open source world, and find a more scalable method that works asynchronously.

1.3 Workspace Structure and Builds

The workspace is divided into a number of directories. The current structure will change somewhat after I finish work on another putback that restructures the workspace to simplify its organization and remove obsolete junk.

Directories	Purpose	Coming Changes
ant	contains version of ant we use	
deleted_files	contains removed files for TeamWare	Not present in CVS
build	created during build process	
docs	contains ORB docs	will eventually contain all ORB docs (moves to www in CVS)
freezept	snapshots of ORB for app server releases	Only in TeamWare, not in CVS
lib	jar files needed to build ORB	
experimental	(coming)	contains experimental code not delivered anywhere at present (and not present in CVS)
make	contains ant build file, gnumake files	
nbproject	contains NetBeans support to treat workspace as a project	
optional	source files that go into optorbcomp.jar (not JDK)	experimental code moves to experimental
src	main ORB source code	
test	main ORB test source	test12, values, and values2 are moved under here
test12, values, values2	more ORB tests for the IBM test suite	contents moved under test
tools	IOR parser	eventually pull into main source code

Note that deleted_files has gotten rather large: it contains about half the number of files in the ORB. Generally when you are using the workspace, if you do not plan to delete any files, you don't need to bring over deleted_files.

1.3.1 docs

There are quite a few files here. This will eventually include all of the useful files from /java/j2ee/CORBA. Most of the doc files in the current workspace are useless, with one important exception: the eal directory.

These are Everett Anderson's notes on the design and implementation of the GIOP layer, and are still quite useful. This also documents the RMI-IIOP stream format version 2 implementation.

1.3.2 ORB source

Generally, most ORB implementation source packages have two parts: an interface defined in an spi package, and an implementation in an impl package. We will generally discuss these together.

The ORB source code is divided into several parts:

src/solaris Useless and should be removed.

src/share/classes This contains the main part of the source code. It can be further divided into:

org OMG standard CORBA APIs

javax OMG standard RMI-IIOP APIs

sun/rmi rmic compiler

sun/corba CORBA Bridge class, which isolates ORB dependencies on non-public JDK APIs

com/sun/corba/se Main body of ORB source code (automatically renamed to ee package for app server build)

GiopIDL IDL files for GIOP protocol definitions

PortableActivationIDL Old ideas for ORBD rewrite; to be removed

experimental/portableactivation Part of ORBD rewrite; to be removed

impl/spi The main part of the ORB implementation

activation ORBD

copyobject Object copier code (not the fast version)

corba(impl) Implementation of OMG CORBA APIs (but not org.omg.CORBA.ORB)

dynamicany Dynamic Any support

encoding Input and output streams for CDR and JSG (Java Serialization for GIOP: Ram's work, conceptually similar to BEA T3). PEPT encoding level code.

extension(spi) Some AS-specific CORBA policies used by EJB for creating POAs.

interceptors(impl) Portable Interceptor implementation

io(impl) Most of the hard part of GIOP (closely related to Java serialization, but not JSG)

ior How we represent IORs

javax(impl) RMI-IIOP implementation

legacy Some AS-specific extensions to interceptors and some connection management support

logging Logging infrastructure used by generated log wrappers

monitoring ORB monitoring framework

oa Object Adapters. This includes:

poa(impl) The Portable Object Adapter implementation

toa(impl) Simple OA used for orb.connect/disconnect (JDK 1.2) and old RMI-IIOP support

rfm The ReferenceFactoryManager, used to enable suspend/resume of ORB processing for dynamic reconfiguration. Used to support dynamic FOLB in AS9.

***.java** The OA SPI.

orb The implementation of the ORB class, and associated configuration framework. The configuration code should really be moved into a utility library.

orbutil Utilities not related to RMI-IIOP. This includes:

closure Simple closure support used in a few places

fsm Finite State Machine library used in the POA to support ActiveObjectMap entry semantics. Should be more widely used in ORB (e.g. connection management). Supports much of the UML state model (but not nested states, and not petri-net style operations)

generic Some useful utilities related to Java 5 generic (generic Pair, various kinds of generic function classes)

proxy Utilities related to simplify construction of InvocationHandlers for java.lang.reflect.Proxy.

template Some sketches of ideas for a general template mechanism I worked on a while ago. Should either be experimental or deleted (generally I'd use codegen now)

threadpool The ORB threadpool implementation that is shared with the app server (should revisit this in light of JDK 5 executors).

timer ORB timing facility

.java(impl) There are a large number of Java classes defined here as well. ORBConstants is very commonly used in the ORB and App server and should be moved to an SPI package.

presentation PEPt presentation level code

protocol PEPt protocol level code

resolver Internal classes used to support string to object reference conversion

servicecontext Internal representation of GIOP ServiceContexts.

transport PEPt transport level code

txpoa TSIdentificationImpl, which is used to connect the ORB and the transaction service.

util low-level code mostly related to RMI-IIOP.

internal Old (JDK 1.4 and earlier) ORB and JNI library related classes that we maintain for backward compatibility.

org/omg Various mostly CSIV2 related protocol definitions (we compile the CSIV2 IDL in the ORB so that the app server CSIV2 implementation can use it).

pept The PEPt 1.0 code used in the ORB.

com/sun/org/omg Some not quite standard OMG classes that were still in flux in the CORBA 2.4.1 timeframe (these are internal only, so this doesn't matter much now)

com/sun/tools/corba/se A number of tools:

- idl** The idl compiler
- jmk** The tool used to validate .jmk files against the contents of the source directories
- logutil** The source code for the jschemeutil.jar library

optional/src/share/classes/com/sun/corba/se The optional classes used only in the app server

org Contains the ASM and BCEL code used in the ORB (and renamed for delivery into the app server). Note that I plan to remove the BCEL code in the near future.

impl/spi Contains a number of modules:

- codegen** The runtime byte code generator library.
- copyobject** Two versions of the reflective copyobject code.
- folb** The AS9 IIOP failover and loadbalancing code.
- relation(EXPERIMENTAL)** Complete support for binary relations (in the mathematical sense) as first class Java objects.
- component(EXPERIMENTAL)** My current work on a Java component system
- plugin(impl.only)** ORB Hardware loadbalancing support
- presentation(impl.only)** fast copyobject code

Note: the EXPERIMENTAL packages will be moved from optional to experimental in the future.

1.3.3 ORB Renaming

Most of the ORB is packaged under the `com.sun.corba.se` package. A version of the ORB code exists in this package (or in a slightly different version) in every JDK since 1.2. We also deliver the ORB code to the Sun application server (now project GlassFish). To avoid possibly collisions between the classes in the JDK and the classes in GlassFish, we rename all the files in the ORB to the `com.sun.corba.ee` package.

This rename is done automatically using a collection of scripts and makefile targets (TBD: make this work purely from ant). Until very recently, it was necessary to rename the workspace before building or running tests. The rename is incremental and fairly quick for small changes (IF you are using a fast local file system), but takes several minutes for a fresh copy of the source. Of course, the rename also interferes with the standard edit-compile-test-debug cycle. I have a build environment built around vim that gets around this problem, but it is difficult to impossible to deal with renaming with any IDE that I know of.

The rename is no longer necessary for development in GlassFish-CORBA. Here all that is required is an ant build, from NetBeans, standalone, or from any other tool a developer might care to use. The only time a rename is required is when the jar files are created that are delivered to GlassFish for integration in the app server.

1.3.4 ORB Build Files

The ORB can be built either with ant or gnumake. We may at some point remove all makefile support, if we can use the Ant build in the JDK. For now, both build systems must be maintained.

1.3.4.1 Building the ORB

The ORB can only be fully built using gnumake, but most of the build can also be done with ant, which is generally what we prefer today. The ORB builds as part of the JDK, so here we just focus on the app server specific build. You MUST use JDK 5 or later.

Assume `<ws>` is the ORB workspace. The basic build sequence is:

1. `cd <ws>/make`
2. `gnumake -f Makefile.corba renameonly`
3. `cd <ws>/build/rename/ee/make`
4. `gnumake -f Makefile.corba buildall` (debugall for debug build, or just set `DEBUG_CLASSFILES` to true)
5. `gnumake -f Makefile.corba buildoptional`
6. `gnumake -f Makefile.corba releaseforappserveronly` (for AS delivery: not needed for testing)
7. `gnumake -f Makefile.corba verifyall` (to run all tests)

An alternative sequence using ant explicitly:

1. `cd <ws>/make`
2. `gnumake -f Makefile.corba renameonly`
3. `cd <ws>/build/rename/ee/make`
4. `ant build`
5. `gnumake -f Makefile.corba verifyall` (to run all tests) (or `ant test`, in GlassFish-CORBA)

I generally use a script to automate this. It is also possible to do a build from NetBeans. It used to be necessary to rename in order to build and run the tests, but that has been fixed in the GlassFish-CORBA project.

Another small note: it is possible to generate JavaDocs for the CORBA SPI. To do this, simply go to the make directory in the renamed version of the workspace, and run “ant javadoc”. The resulting JavaDocs may then be accessed from <ws>/build/renamed/ee/build/release/docs/index.html (exercise for the reader: fix all of the JavaDoc warnings that show up). It should also be possible to create the SPI javadocs without renaming, and the NetBean project supports this as well.

1.3.4.2 Structure of the build files

Let’s look at the ant files first. There are several .xml files, all included in build.xml:

build.xml the main file

jscheme.xml Ant targets for running jscheme and generating log wrappers

src-idl.xml Ant targets for generating java from idl for the main ORB code

test-idl.xml Ant targets for generating java from idl for the ORB tests

test-rmic.xml Ant targets for generating stub and skeletons using rmic for the ORB tests

test.xml Ant targets for running the ORB tests (currently only in GlassFish-CORBA)

Support for gnumake includes:

Makefile.corba The main makefile

com,sun,org Trees containing makefiles for all of the ORB packages. This partly mirrors the structure of the source packages, but unfortunately the correspondence is not exact.

common Other makefiles included in Makefile.corba

mininclude All of the .jmk files. These EXACTLY reflect the structure of the source code, but must be maintained by hand. These files are essential for the operation of the gnumake build. The contents are checked against the actual source files on each build, and any discrepancies are treated as errors. The .jmk files must be updated every time a Java source file is added, deleted, or moved in the src/share/classes packages. Note that the tests and the optional classes are not included here. This added complexity exists solely to support building CORBA as part of the JDK, and will be revisited in the future.

1.3.5 tests

The ORB has many tests in several different test suites (docs/TestCases.sxc gives some details of the contents and the numbers of test cases in the test suites, from a rough manual count). The test suites are:

ibm Old RMI-IIOP tests created by IBM. These are the lowest quality, hardest tests to deal with.

corba The bulk of our developer tests. Covers all areas of the ORB to some degree.

pi The Portable Interceptor tests.

naming Test for the name services.

mantis Tests specifically for bug fixes made to JDK 1.4.1.

hopper Tests specifically for bug fixes made to JDK 1.4.2.

copyobject Tests for the various object copiers. Can also be used as a timing test for streams, which will likely be very important to us.

simpleperf A simple performance test mainly for colocated calls.

1.3.6 libraries

There are a number of libraries in the lib directory. These are all used either for building or testing.

ejb-2_1-api.jar Needed for codegen test

ir.idl Old and obsolete interface repository file (should be deleted)

javatest.jar Old version of Sun's JavaTest framework: not currently used; should probably be deleted

jcov.jar Used for gathering coverage information during test runs when that option is enabled

jdom.jar JDom implementation used in (I think) Harold's generic RPC message test (XML-RPC in ORB framework)

jmxri.jar An early version of JMX (pre-JDK 5) used in a test: may be able to get rid of this

jscheme.jar Scheme interpreter used for generating log wrapper source files

jschemelogutil.jar Some simple utilities used with JScheme (source is in the workspace)

junit.jar JUnit, used for some of the ORB tests

kawa.jar Another Scheme in Java implementation (called kawa) that is also used for XQuery processing, which I used once for generating a new CORBA schedule from the XML version of the jxproject schedule

orb.idl A standard IDL file for some standard definitions (not really used)

rt.idl IDL for the CodeBase interface (part of the SendingContext module; used to enable access to another VM's typing information for RMI-IIOP)

xalan.jar Apache xalan, used in a test

xerces.jar Apache xerces, used in a test

1.4 ORB Developer Tests

1.4.1 Test Suites

The ORB developer tests are found in the directories test and optional/test. The makefile for running the tests is in test/make/Makefile, but the test suites can also be run from make/Makefile.corba. This table gives a brief overview of the available test suites:

Suite Name	Test File	Makefile.corba Target	test/Makefile Target	ant target in GlassFish-CORBA	Purpose
all	-	verifyall		test	runs all test suites
IBM	test/ AllTests.txt	verifyonly	verifybuild	test-rmi-iiop	Old IBM tests
corba	corba/ CORBATests.txt	verifyonly_ corba	verifybuild_ corba	test-corba	Most of the newer tests
pi	pi/ PITests.txt	verifyonly_ pi	verifybuild_ pi	test-pi	Portable Interceptors tests
copyobject	corba/ CopyObjectTests.txt	verifyonly_ copyobject	verifybuild_ copyobject	test-copyobject	copyobject test
naming	naming/ NamingTests.txt	verifyonly_ naming	verifybuild_ naming	test-naming	naming tests
hopper	hopper/ HopperTests.txt	verifyonly_ hopper	verifybuild_ hopper	test-hopper	Bugfixes for JDK 1.4.1
mantis	mantis/ MantisTests.txt	verifyonly_ mantis	verifybuild_ mantis	test-mantis	Bugfixes for JDK 1.4.2
simpleperf	performance/ Tests.txt	verifyonly_ perftest	perftest	test-perf	co-located call performance test

All of the tests can be run either from `<ws>/make` or from `<ws>/test/make`, using the appropriate targets. The output of the tests (stdout and stderr) are redirected to log files. These log files are located in `<ws>/test/make/gen`, under the package name of the individual tests in the test suite. Note that running a test suite first empties `<ws>/test/make/gen`, so you cannot currently run all test suites and look at all failures. However, most of the tests of interest are in the corba test suite, so making those pass first is usually all that is needed. This is not always the case, particularly if you are working on the RMI-IIOP code (which is tested in the IBM test suite). These tests are also described briefly in `<ws>/docs/TestCases.sxc`, which is a spreadsheet that roughly counts the number of test cases in each test in each test suite.

Note that the names of the target will change in the near future. The tests will also be made to run from ant, which is essential to fully utilizing NetBeans for ORB development.

Each of the tests in the test suite starts one or more Controllers. A Controller is simply a class that controls a component of a test. For example, many of the tests have 3 controllers: a Client, a Server, and ORBD (which is used mainly for name service, as few tests actually exercise server activation).

Adding a new test is simple: just create a new package for testing, write the test, and add it to the appropriate test file. There is a document that is somewhat helpful in the workspace at

`test/src/share/classes/corba/framework/package.html`

Especially read the section on the Controller classes, as that is really the heart of the test framework. However, the document is old, and somewhat out of date. You should consider the following additions and changes since the document was written:

- You can use JUnit for tests. Simple write a JUnit test, and then wrap it with a simple test class that extends `corba.framework.CORBATest`. In fact, ANY program can be included in the CORBA test framework this way. All that is needed is for the embedded test to indicate success by returning 0, and failure by return a positive value in a `System.exit()` call.

The following tests currently use JUnit:

- `corba/copyobject` (this is the most complex example of what can be done with JUnit)
- `corba/dynamicrmiiop`
- `corba/stubserialization`
- `corba/misc` (this is a good test to use a simple example)
- `corba/messagetrace`

- Similarly, you can also use TestNG for tests. The corba/timer and corba/mixedorb tests both use TestNG.
- Makefiles are not needed. The ant build takes care of everything. However, the ant build needs to be updated if new tests that require IDL or RMIC are needed.
- You can write tests that use RMI-IIOP without needing rmic. To do this, just use dynamic RMI-IIOP. For an example of how to do this, look at the corba/rfm test. Basically you just need to use the PresentationManager API for a couple of things (access to the repository ID and to create a Tie), and you also need to make sure that dynamic RMI-IIOP is enabled. A renamed test will automatically run under dynamic RMI-IIOP, but if the test is NOT renamed (and this will be the case after the migration to GlassFish, at least for pure ORB development), you need to add the following static initializer to the test code:

```
static {
    System.setProperty( ORBConstants.USE_DYNAMIC_STUB_PROPERTY, "true" );
}
```

- Conversion status info is out of date in the document: ignore this section.
- Debugging is better. The document mentions the RDebugExec controller (and also ODebugExec, but the omniscient debugger has not been tested or updated in years. See <http://www.lambdacs.com/debugger/debugger.htm> for more information).
- Default ORB class has changed. We now use com.sun.corba.se.impl.orb.ORBImpl.
- There are two security policy files available for running the tests. The default test.policy file simply sets up the needed permissions for all of the tests and the ORB. The more fine grained test.policy.secure file sets up more restrictive permissions for the tests, while giving more powerful permissions to the ORB code in the build and optional/build directories. Which policy file is used is set in the test/make/Makefile in the DEFINES macro.

Setting the more secure policy file is useful to work on ensuring that the ORB has doPrivileged blocks around all operations that have security implications. It is known that the ORB is at least somewhat deficient in this area, but we have not taken the time to thoroughly address this issue.

- There are several environment variables that are useful to set while running tests (see test/make/Makefile for more details):

STATIC_STUB when set to 1, forces the use of static RMI-IIOP

DYNAMIC_STUB when set to 1, forces the use of dynamic RMI-IIOP

BCEL_COPYOBJECT when set to 1, uses the BCEL version of the fast object copier (experimental)

JAVA_SERIALIZATION when set to 0, use CDR instead of JSG (JSG is experimental, but enabled by default in the workspace only)

DEBUGGER can be set as follows:

- 1 Set ORBDebugForkedProcess=true (for IBM tests), and run tests so that a JPDA-compliant debugger can be attached
- 2 Run tests under OptimizeIt
- 3 Run tests withc -Djcov=true for coverage analysis

1.4.2 debugging tests

To debug a test, you need to know the name of the controller(s) to which you need to attach a debugger. Controllers are normally created by the methods `createORBD`, `createClient`, and `createServer`. The default names of the controllers are `ORBD`, `Client`, and `Server`, respectively. The `createClient` and `createServer` methods can also take a second argument (the first is the class name of the test program) that gives a specific name for the controller.

Given the name of the controller(s) to debug, simply add the argument

```
-rdebug XXX,YYY
```

for controllers `XXX` and `YYY` (for example) to the end of the test file argument that starts the test.

ORB debug flags can also be passed into a test. To do this, add the argument

```
-orbtrace XXX:f1,f2;YYY:f3
```

where the argument is a semi-colon separated. Each element of this list starts with a controller name, followed by a comma separated list of ORB debug flag names (see `com.sun.corba.se.spi.orb.ORB` for the current list).

It is also possible to change the log levels so that ORB log information can be displayed on the console (or anywhere else, depending on the log system configuration). This follows the usual log system mechanisms. The ORB logger names are discussed in 8.4 on page 33.

1.5 ORB SQE Tests

Sony Manuel maintains a large collection of CORBA SQE tests in `/java/idl/ws/rip/RIP_TEST_MASTER`. Read the file `DTF_RTM_README.html` in this workspace for details (and contact Sony as well). These test are not currently available in GlassFish-CORBA.

We should look at creating a tighter integration between the CORBA dev tests and the SQE tests at some point. In particular, I'd like to have the POA and INS tests run automatically as part of the CORBA dev test cycle.

1.6 Thoughts on Middleware goals

Middleware is a rather complex kind of software to build well. It spans many parts of the computer science discipline, including compilers, operating systems, and network communications. Middleware tends to be complex, long-lived software, and there are many different ways to build it. Our goal in developing the ORB has been to develop very flexible and high-performance middleware while maintaining a clear (if complex) architecture that can be easily composed, ultimately out of re-usable modules. One way to look at an ORB is that `ORB.init` creates a particular middleware implementation that is specialized to the needs of an application. For example, the behavior of the default ORB in the JDK is rather different from that of the ORB in the app server, even though both share >95% of the same code.

At least the following dimensions must be considered in order to build effective middleware:

Flexibility. Middleware has been changing constantly since Nelson invented RPC around 1980 or so. Much of Harold's work on PEPT has been devoted to dealing effectively with this aspect of middleware construction. The following elements often vary indendently of each other:

Presentation. By this we mean the kinds of data types that may be passed between a client and server (roles here, as any given software entity often acts in both roles), and the APIs and other structures used to collect these kinds of data together.

Examples of data types include the IDL data model from CORBA, the Java data model from RMI, XML schema, ASN.1, SUN RPC, MIME types, and many others.

There are broadly speaking two ways to view the API question: either the system uses some sort of Proxy to make a remote call "look like" a call to an abstraction (method call, procedure

call, SmallTalk message send, etc) or the API provides an explicit representation of a request or message that a program can set up with data and then send (CORBA dynamic invocation, Message Oriented Middleware, and others).

Encoding. Given a structure containing the data from the presentation layer, it needs to be converted from an in memory representation to some representation suitable for transmitting across some serial medium like a TCP connection. This is usually referred to as marshalling and unmarshalling the data, and is often the most expensive operation that middleware performs. In fact, I believe this is the most significant challenge for middleware of all types as network hardware increases in speed.

Protocol. This is related to but distinct from encoding. Here we are concerned about the kinds of messages that clients and servers exchange. This is mainly about message framing, headers, various kinds of meta-data associated with requests, and the distributed state machines involved in protocol design.

Transport. Ultimately, some mechanism must be used to transmit data from the client to the server. This can include network protocols, shared memory mechanisms, Solaris doors, and direct calls within the same address space that bypass all such mechanisms (in which case the protocol and encoding are much simplified as well).

Performance. Everyone always wants middleware to perform as quickly as possible. This is quite a challenge. First, there are many aspects to performance, including:

Latency. How long does it take to send messages of various types? What about short vs. very long messages?

Throughput. How many request per second can be sent through the system? This often conflicts with latency. For example, a front end concentrator can help to pump more data through, at the cost of increased latency due to any extra hop (e.g. NTTWest's use of a hardware load balancer).

Creating Endpoints. Object-Oriented remoting systems like CORBA create, marshal, and unmarshal endpoints (IORs in CORBA) constantly, and the performance of such operations is quite significant.

Today, our latency is poor (except for co-located RMI-IIOP calls, which are highly optimized), our throughput is OK (and this is NIO select related), and the IOR handling is pretty good, although much more is possible.

We need to follow a number of strategies for improving performance:

- Cache when appropriate. The ORB caches lots of information, especially related to class-specific information for marshalling and handling stubs and skeletons.
- Don't do it if you don't need it. For example, interceptors are expensive, so don't pay overhead for them if none are installed. A bad example is that the encoding currently computes two indices instead of one, and checks to see whether to marshal little- or big-endian on every primitive marshalling operation. These are areas that we should investigate soon.

This is particularly important for systems that have extreme flexibility.

- Precompute where possible (really a form of caching). A good example of this is the use of IOR templates to make endpoint (IOR) construction very fast. Creating a POA creates an IOR template, and creating an IOR from an IOR template is almost a trivial construction.
- Avoid data copying. This is one of the main areas of important for transport optimization. For example, this is where use of direct ByteBuffers can help performance (which we do today).
- Consider runtime code generation. We have started to do this already for dynamic RMI-IIOP. I also have been experimenting with doing this for fast object copying, and marshalling is another candidate.

- Instrument the code so that performance can be understood. We have started to add internal instrumentation for this, and I think we will expand on this approach. Sometime the most appropriate tool is an external tool like OptimizeIt, but both approaches have their advantages. Note that applying OptimizeIt to an ORB test is easy: see 1.4.1 on page 7.

Reliability. Middleware is usually used in places that need to run continuously (like the app server). This again implies a number of considerations:

Avoid Memory Leaks. This has been an issue for the ORB mainly in caching class related data. We have had to make careful use of soft and weak references in a number of places to handle this. Extreme care is needed to avoid leaking direct ByteBuffers, which generally must be pooled to achieve acceptable performance.

Build Clean Code. This is why we have a strong emphasis on programming to interfaces. Another important aspect of this is only write a piece of code once, and then reuse it.

Use Test Driven Development. By this I mean that anyone producing a module of code must also produce a set of unit tests (and other tests as needed) to validate the correctness of the module. This also means that as much as possible, bug fixes require a test that fails, and a fix that makes the test pass, and the test must be incorporated into the automated build.

1.7 ORB Coding Practices

- Use comments wisely. Some guidelines:
 - Don't document the obvious. If the code is clear, it should speak for itself. If the code is not clear, try to make it clear. If you can't, then comment about its operation.
 - Try to choose meaningful names to reduce the need for comments.
 - DO document global issues that span more than one method/class/package. These are the most important comments to include.
 - Always document public methods (there should be NO public data members, except possibly for some static constants). Follow the standards for using JavaDoc.
- Program to interfaces. Most of the ORB is built this way. There are a number of characteristics of this design:
 - Use of the factory pattern. Factories in the ORB are generally classes in spi packages which contain only static methods for accessing factories, or standard instances of particular interfaces. These classes should end in the name "Default(s)" or "Factory".
 - **new** is only used to create instances of interfaces inside the factory classes. This makes it much easier to re-use code, and to isolate the client from caching vs. creation decisions.
 - Initialization drives call flow. That is, in order to understand what really happens at runtime, you must know how the ORB initialization set up the concrete instances behind the interfaces. Similarly, in order to understand the dispatch cycle, you must understand how the IOR is created and marshalled (especially if we ever do a Solaris doors transport again).
- Avoid magic strings and numbers. If you need a constant, put it in ORBConstants, since it will either need to be configurable or else referencable in more than one place in most cases.
- Generally use only SPI classes (but note ORBConstants is in impl). Generally an ORB implementation class is free to use any required spi class, but should avoid using implementation classes outside of its own package.
- The ORB object instance is central. It is the repository for all runtime ORB data. It drives the configuration of the runtime ORB. Everything that the ORB provides is accessible from the `com.sun.corba.se.spi.orb.ORB` class.

- Access the ORB either as `org.omg.CORBA.ORB` or as `com.sun.corba.se.spi.ORB`. There are other ORB classes, but never program to them.
- Use the CORBA ORB for code that may be shared with other ORB implementations. This is mainly an issue in the RMI-IIOP code and the value handler, which are used by some third party ORBs. If you need the ORB SPI, provide code that handles the CORBA ORB case as well.
- ORB extensions should be created as IDL local interfaces, and made accessible through a `resolve_initial_references` call. We have not followed this principal in the past as much as we should have, and the result is that there are too many methods in `spi.ORB`. An alternative here is to use dependency injection from a component framework, but that's a separate topic (and paper).
- Read “Effective Java” (if you haven't already). Josh Bloch's book is full of some very good advice that should almost always be followed.
- Try to avoid complex constructors, or classes with many constructors. This makes the code confusing and hard to use.
- Try to complete instance initialization in the constructor. If this is not possible, check whether the instance is in a good state before continuing with the body of a method.
- Some ideas on error handling:
 - Design code to fail fast. That is, if there is a problem in the code, fail sooner, so that the problem is correctly reported, rather than letting it propagate through the system until it's hard to determine what happened. Examples of this include use of **assert**, and the ORB's use of `INTERNAL SystemExceptions` to report consistency failures.
 - Use exception chaining when reporting errors. The log wrappers make this easy to do.
 - Use system exceptions to report errors. We have a pretty good log wrapper mechanism that makes this easy to do: always use it. The only exception here is in ORB independent libraries (like codegen). Here, just use standard Java exceptions.
 - Prefer unchecked exceptions in most cases. This has been a raging debate for a long time in the Java community. I think checked exceptions are occasionally useful, but often more trouble than they are worth. A good rule of thumb is that if the only thing an application is going to do is pass the exception on to another layer, it should be unchecked. Writing code that catches a series of checked exceptions and handles each through a standard reporting mechanism is wasteful and annoying, and does nothing for the readability of the code.
- Avoid import `xxx.*`. Careful organization of import lists is a great aid to figuring out how classes are coupled together. The ORB rules that should be followed for import lists are:
 1. Arrange packages from the general to the specific. For example, put `java.*` first, then `org.omg.*` first, finally internal `com.sun.corba.se.spi.*` classes.
 2. Within a package, arrange classes in alphabetical order.
 3. Separate different package imports with a blank line.
 4. If you have an IDE that can do this for you, use it.
- Be very careful with import static `xxx.*`. It is occasionally very useful (for example, for calling all the static methods like `_class`, `_method`, `_if` etc. in the codegen library). But using it for all statics could lead to much confusion.
- Names are very important
 - Interfaces have descriptive noun phrases (e.g. `LocalClientRequestDispatcher`)
 - Implementations of interfaces end in “Impl” (e.g. `JIDLLocalCRDImpl`)

- Try to avoid abbreviations (but not if the name is too long)
 - Limit names to 7 nouns (and I think we break this one once or twice)
 - Multiple implementations should look like <InterfaceName><Characteristic>Impl or <Characteristic><InterfaceName>
 - Abstract base classes should look like <InterfaceName>Base
 - Factory interfaces usually look like <InterfaceName>Factory
 - Follow the standard Java naming conventions (except for IDL generated methods)
- Prefer short methods (and let HotSpot do its job). Some of the code in the ORB (e.g. CDR streams) has badly violated this rule, resulting in many cut-and-paste sections of nearly identical code. This is a testing, debugging, and understanding nightmare.
 - Pay attention to cohesion and coupling. A class should do one thing that can be crisply articulated, as should a method. A class should use a minimum of other classes to get its job done.
 - Either prevent inheritance or design for it. Harold and I have often been in conflict on this one. There is some code in the ORB in which the data members are private instead of protected. This is occasionally helpful (for example, in some of the HWLB plugin code), but is an open door to maintenance headaches. If a field or method is protected, you are really saying that no invoker can call this, but a subclass can, and a subclass can override this method. What are the constraints on subclasses in this case? Such design decisions should be documented in the code.
 - Use shallow inheritance hierarchies. We commonly use the interface/abstract base class/several concrete classes pattern to facilitate code reuse across implementation variations. There are a few places (e.g. local client request dispatchers, some of the POAPolicyMediator code) that do this for two (perhaps more?) layers, but that should be rare.
 - Don't bother making methods final for optimization, as this is not helpful. Final is semantic: it means that no subclass can override this method.
 - Utility classes (those that only contain static methods) must be final and have an empty private no-args constructor. This prevents all instantiation and subclasses of utility classes.
 - Consider making most local data final. This is unusual, but actually quite useful, since most local data is initialized and then referenced in the method. Parameters should always be final, as assigning to a parameter is poor style.
 - Never use public data members in classes (except for static final). Be careful with protected and default access for data members.
 - Avoid non-constant static data members in classes. Sometimes they are necessary, in which case concurrency protection is usually required.
 - Use JDK 5 (and convert old code to use it). The new language extensions help to make for more readable code. I have converted some of the code, but not all.
 - Design for testability. Make sure that classes have an interface that supports testability: if some internal state is established that needs to be verified, then there must be an interface to access the state. Leaving code for testing in the product code is acceptable, but make sure it does not affect any critical performance issues.
 - Design for concurrency.
 - Make sure you know whether instances of a class may be used from more than one thread concurrently. If so, use locking.
 - NEVER assume that it is safe to access ANY data without either locking or declaring it to be volatile. Don't break this rule for stats counters under the assumption that it will only be off a little (this isn't true in complex SMP servers with large caches).

- Make use of the Java 5 concurrency utilities. These are very well designed to solve a number of hard problems.
- If you need to wait on more than one condition, use the `java.concurrent.util.locks` package, particularly `ReentrantLock` and `Condition`.
- Use a lock hierarchy to avoid deadlocks if multiple locks are required in a thread of control.
- Be aware of the potential for locking hot spots on Sun's SMP servers (and especially Niagra these days). This frequently is a single lock that every request dispatched through the ORB must acquire at some point. Again, Java 5 has some very useful utilities to help (such as `ConcurrentHashMap`).
- Do NOT put Class instance into Maps as values or keys directly or indirectly. Pinning a Class object in a Map will prevent the Class from being garbage collected, which will prevent that Class's `ClassLoader`, and all other Classes loaded by the `ClassLoader` from being garbage collected. This has led to massive memory leaks in the App Server on several occasions.

The solution here is to use `WeakHashMap` (for Classes in keys) or `SoftCache` (for Classes in values) as needed. Grep the source code for examples.

1.8 Supporting JDK and App Server

The main body of the ORB code (that is, most of the contents of the `src` directory) must be delivered into both the JDK and the app server without change (other than the automatic rename). This has some interesting implications:

1. Use extension, not variation. Never create an app server and a JDK version of the same class. Instead, refactor the class so that there is a common base in the core that can be extended, either through inheritance or composition.
2. Prefer composition to inheritance.
3. Remember that the `io`, `util`, and `javax` code is shared between our ORB and other (non-Sun) ORBs. This means in particular that other ORBs may be built using our `ValueHandler` and `RMI-IIOP` classes. The main issue here is that we cannot assume that the ORB class is always `com.sun.corba.se.spi.orb.ORB`. It is OK to create a specialized path for our ORB, but this must always be done with an **instanceof** check, and handle the non-Sun ORB case correctly.
4. All behavior exhibited by the JDK ORB must follow the OMG specifications. Mostly this is CORBA 2.3.1, but we follow the semantics of later versions when errors have been corrected. For example, the POA should basically follow the behavior documented for CORBA 3.0 at this point. Note that it is fine to create non-standard extensions to CORBA semantics for the app server: this is exactly what we have done for failover and load balancing support (among others).

Chapter 2

The ORB Class

The ORB class is the central control point in the Sun ORB implementation. Here we will examine its structure, the services it provides, and how it is initialized and terminated.

2.1 Inheritance Structure

The ORB class has the longest inheritance chain in the ORB. Constructing a UML diagram for this is difficult (because of the way this works in Java Studio Enterprise) and not very illuminating. Instead, I'll just list the classes and their main function:

org.omg.CORBA.ORB: This is the main ORB API defined by the OMG.

org.omg.CORBA_2_3.ORB: This adds some methods related to value types.

com.sun.corba.se.org.omg.CORBA.ORB: This adds `register_initial_reference`. While the method is an OMG standard, the Java mapping that we used for the ORB API did not include this method, so we put it here.

com.sun.corba.se.spi.orb.ORB: This adds all of the internal SPI methods to the ORB.

com.sun.corba.se.impl.orb.ORBImpl: This is the ORB implementation.

com.sun.corba.se.internal.iiop.ORB: For backwards compatibility with JDK 1.4.

com.sun.corba.se.internal.POAORB: For backwards compatibility with JDK 1.4.

com.sun.corba.se.internal.PIORB: For backwards compatibility with JDK 1.4.

The first 3 classes together define the standard OMG ORB API. It is split because it evolved in stages, and we cannot add new methods to an interface that a third party may implement. Instead, we extend the API class, allowing older ORB implementations from third parties to continue to work. The classes in the internal package can be ignored. They are present so that any old code written with the `ORBClass` property set to the class name will continue to work.

2.2 ORB SPI structure

The key part of the ORB SPI is in the `com.sun.corba.se.spi.orb.ORB` abstract class. This is an abstract class because it must extend the other ORB abstract API classes defined by the OMG. The ORB class provides the following operations:

- A set of debug flags. ORB defines a number of public boolean data members of the form `xxxDebugFlag`. These can be set to true using the `ORBConstants.DEBUG_PROPERTY` property. Adding new flags is simple: just follow the existing pattern. These flags are accessible to any part of the ORB that uses an ORB class instance, which is everything except RMI-IIOP and some libraries.

- Methods for testing for local host and server id, which is important for determining when a call is colocated. This mechanism should be revisited in order to deal with multi-homed hosts and calls between different ORB instances (which currently are not optimized, even if the ORB instances are in the same VM).
- Methods for manipulating the OAIInvocationInfo stack. This stack contains information about the current request on the server side. The ORB has several stacks that serve similar functions, as does the EJB layer in the app server. This suggests an important optimization: unify the lifecycle management of the stacks, and use a single unified stack on each side (client and server) that supports extensible data elements.
- Access to a number of managers, factories, and registries:
 - CorbaTransportManager
 - LegacyServerSocketManager
 - PresentationManager (this is a static, as it is shared across RMI-IIOP and all ORB instances).
 - PresentationManager.StubFactoryFactory (also static)
 - MonitoringManager
 - PIHandler, which provides all of the methods needed to support Portable Interceptors.
 - ServiceContextFactoryRegistry
 - RequestDispatcherRegistry, which provides many of the key objects needed for the dispatch cycle.
 - ORBData, which contains all of the ORB configuration data.
 - ClientDelegateFactory, which converts a CorbaContactInfoList into a CorbaClientDelegate.
 - CorbaContactInfoListFactory, which converts an IOR into a CorbaContactInfoList. The ClientDelegateFactory and the CorbaContactInfoListFactory are the two essential objects used to prepare an endpoint (represented by an object reference, and containing an IOR) into a form suitable for use in the dispatch cycle.
 - Resolvers for resolve_initial_references, list_initial_services, and register_initial_reference. These are also used for string_to_object.
 - Factories and related objects for IOR handling:
 - TaggedComponentFactoryFinder
 - IdentifiableFactoryFinder for TaggedProfiles
 - IdentifiableFactoryFinder for TaggedProfileTemplates
 - ObjectKeyFactory
 - WireObjectKeyTemplate

This allows us to extend the basic IOR framework with tagged component and profiles as needed, and also allows us to plug in a particular object key representation. The object key representation is important for the operation of object adapters.
 - ThreadPoolManager
 - CopierManager
 - ByteBufferPool
- Access to the ORB Invocation Interceptor. This is a non-standard interceptor that intercepts calls before and after the dispatch through the stub (when dynamic RMI-IIOP is used). This is currently used in the app server for call flow analysis, and could also be used for ORB timing points. We may wish to consider providing a more extensible mechanism here: the current implementation supports only one interceptor.
- Access to primitive type codes and a mapping between repository IDs and type codes.

- Access to the current ORB version, and a mechanism to set the ORB version for the current request (to handle interoperability between different versions of the Sun ORB).
- Access to the IOR for the FVD CodeBase object.
- Mechanisms for dealing with bad server IDs. This is only needed to support the ORBD. The mechanism is now obsolete, as the object reference template (ORT) provides a better solution, but our current ORBD has not been updated to use ORT.
- The notifyORB method, which is used to support the DII get_next_response mechanism.
- Support for controlling ORB shutdown so that shutdown cannot happen until all active requests have completed.
- Access to the transient server ID, which is used for IORs for transient object references.
- Methods used by generated log wrapper classes to create log wrappers, which are used for reporting errors and managing CORBA system exceptions.

Clearly there are a lot of methods in the ORB SPI. A better approach would be to push most of this into the initial references mechanism, using IDL local objects. This would substantially reduce the size of the ORB SPI. But I think we are unlikely to do this, as the impact on the existing code is probably too large.

2.3 ORB Initialization

The ORB initialization code is fairly complicated, but reasonably well structured at this point. I'll start with a discussion of the ORB configuration framework, then describe the ORB initialization process in moderate detail. I'll also discuss how this is extended in the app server.

2.3.1 The Configuration Framework

The orb packages contains a mostly general purpose framework for handling configuration. This is divided into several parts:

- The DataCollector, which gathers configuration data from config files, system properties and other properties objects, and command line arguments together into a uniform Property object.
- The Operation interface and factory methods for creating Operation instances. Operation is a simple unary function interface that returns an object. A wide variety of factory methods are provided in OperationFactory and OperationFactoryExt, including one that composes Operations to create a new Operation. This allows expression of fairly complex parsing methods. Many of the Operation instances just convert a String into some other type.
- The PropertyParser and related classes that take a number of operations and combine them into a single parser that can parse all the elements of a Property instance into a Map<String,Object> from field names to values.
- Base classes that use a PropertyParser to initialize (possibly private) fields in a configuration object (typically a JavaBean-like object with only read accessors). There is also a base class (ParserImplTableBase) for constructing a PropertyParser from a table that includes default values and test data. This is used for initializing the ORBData object that contains all of the ORB configuration data.

The following sections will look at these parts in more detail.

2.3.1.1 DataCollector

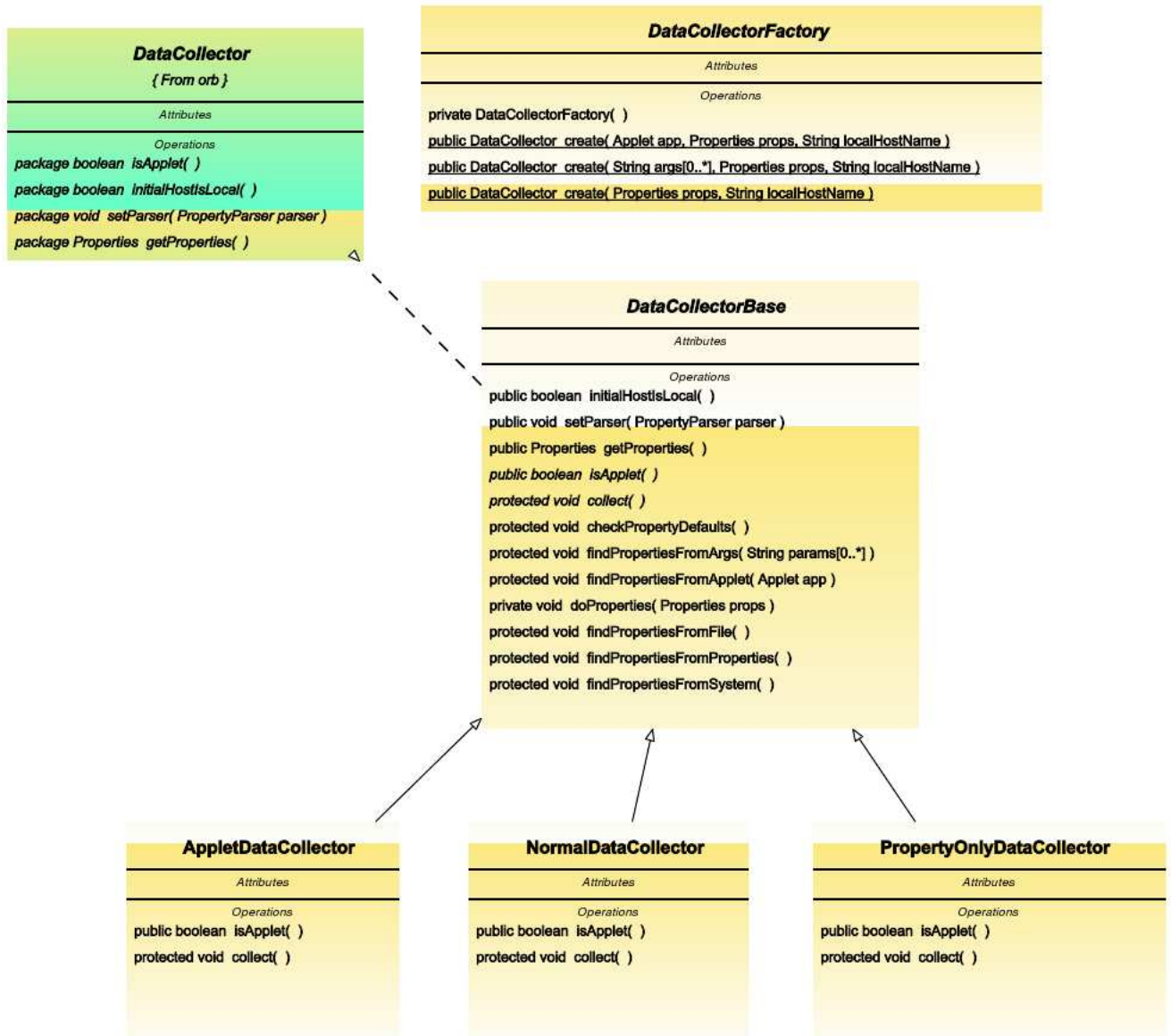
A DataCollector has a rather simple interface: setParser passes a PropertyParser instance to the DataCollector, which causes the DataCollector to gather together all configuration data from the available data sources into a single instance of Properties. This instance is available in the getProperties method.

The available data sources are:

- System Properties
- Applet Properties
- Contents of configuration files
- Command line arguments

It is perhaps not immediately obvious why a PropertyParser needs to be passed to the DataCollector. The reason for this is that it is not always possible to simply grab every bit of information from the data sources. But it is possible to get all configuration information for the known property names. So the DataCollector uses the PropertyParser to fetch information for all property names of interest.

Here is a class diagram of the DataCollector classes:



Note that there are 3 different kinds of DataCollector. The Applet and Normal DataCollectors are used with the corresponding ORB.init methods. The PropertyOnly DataCollector is only used internally, when we need to create an internal full ORB instance to support certain operations on the ORB singleton. The DataCollectorFactory class provides static methods for creating the different kinds of DataCollectors.

There are a few issues here that could be revisited:

- The DataCollector framework has some internal ORB dependencies that could be removed:
 - It handles some URL property names (for applets) specially.
 - There is special handling for -ORBInitRef.
 - The DataCollector base goes to a lot of trouble to hide some sensitive data from arbitrary access by untrusted clients (mainly the local host name). It may be better to avoid doing this, and instead check all access to sensitive information through the SecurityManager (if one is present).
 - While the DataCollector interface is independent of the ORB, DataCollectorBase is not. It would be cleaner to factor the ORB dependencies into another base class.

Fixing these issues would allow reuse of the DataCollector mechanism outside of the ORB, and provide a somewhat cleaner implementation.

2.3.1.2 Operation

The operation interface is simply:

```
public interface Operation {  
    Object operate( Object value ) ;  
}
```

The SPI classes OperationFactory and OperationFactoryExt provide a large number of factory methods for creating instances of the Operation interface:

- makeErrorAction(Operation op), where operate calls op and ignores any errors it may through
- indexAction(int index), where operate returns the index element of its argument, if the argument is an array
- suffixAction, where operate returns the first value in a Pair<String,String> object
- valueAction, where operate returns the second value in a Pair<String,String> object
- identityAction, where operate returns its argument
- booleanAction, where operate converts its argument from a String to a Boolean
- integerAction, where operate converts its argument from a String to an Integer
- stringAction, where operate checks that its argument is a String, and returns it if it is
- classAction, where operate converts its argument from a String into a Class using ORBClassLoader.loadClass
- setFlagAction, where operate returns Boolean.TRUE
- URLAction, where operate converts its argument from a String into a URL
- integerRangeAction(int min, int max), where operate converts its argument from a String to an integer if its argument represents an integer between min and max, otherwise throws an exception
- listAction(String sep, Operation act), where operate expects its argument to be a String of data separated by the sep delimiter, and uses a StringTokenizer to separate the argument into a sequence of Strings. operate then applies act to each String in the sequence, and returns the results in an array. This is used for parsing homogeneous lists of data.
- sequenceAction(String sep, Operation[] act), where operate behaves similarly to listAction, except that successive elements of act are applied to the sequence of Strings, instead of always using the same Operation. This is used to process heterogeneous lists of data.
- composeAction(Operation op1, Operation op2), where operate first applies op1 to its argument, then applies op2 to the result.
- mapAction(Operation op), where operate applies op to each element of its argument, which must be an array, and returns an array of the results.
- mapSequenceAction(Operation[] op), where operate behaves similarly to mapAction, except that successive elements of op are applied to the elements of the argument in sequence.
- convertIntegerToShort, where operate converts an Integer to a Short.

- `convertAction(Class<?> cls)`, where `operate` constructs an instance of `cls` using its argument as an argument to the constructor. Here it is assumed that `cls` contains a constructor that takes a single `String` as an argument.

Other `Operation` implementations can be readily created, but this set is sufficient to handle all ORB configuration parsing (except for URL parsing, which is currently handled by some classes in the resolver package). Extending this framework to handle URL parsing is relatively straightforward, but requires the ability to handle optional data and alternate forms that is not currently present (essentially something like `ifAction(predicate, opTrue, opFalse)` would probably take us in the right direction).

The other issue with this is that composing all of the actions in Java is somewhat cumbersome (take a look at `ParserTable.makeADOperation` for an example). A customized language (e.g. some Lisp macros) could make this much simpler. Combining this with annotation and code generation could reduce the ORB configuration implementation to something like:

```
@Configuration
public interface ORBData {
    @Parse( "<some expression>" )
    public String getORBInitialHost() ;
    ...
}
```

Pursuing this degree of automation is probably more than is justified by the needs of the ORB.

2.3.1.3 PropertyParser

The `DataCollector` gives us a way to gather multiple sources of configuration together into a uniform `Properties` object, and the `Operation` framework gives us a way to parse `Strings` into data in many different ways. The `PropertyParser` ties these two mechanisms together so that we can parse all of the configuration data in a single operation.

A `PropertyParser` is basically a collection of `ParserActions`. There are two kinds of `ParserActions`:

Normal: Here the property name is found in the `Properties` object, and the `String` associated with the name is transformed into a value.

Prefix: Here the property name is a prefix, and all property names that start with the prefix are transformed into the value.

There is a factory class (`ParserActionFactory`) that is used to create the two `ParserActions`.

A `PropertyParser` is initialized by call its `add` and `addPrefix` methods to add the `ParserActions` that are needed in the `PropertyParser`. These methods return the `PropertyParser` so that they can be chained if necessary. Each of these methods takes the following arguments:

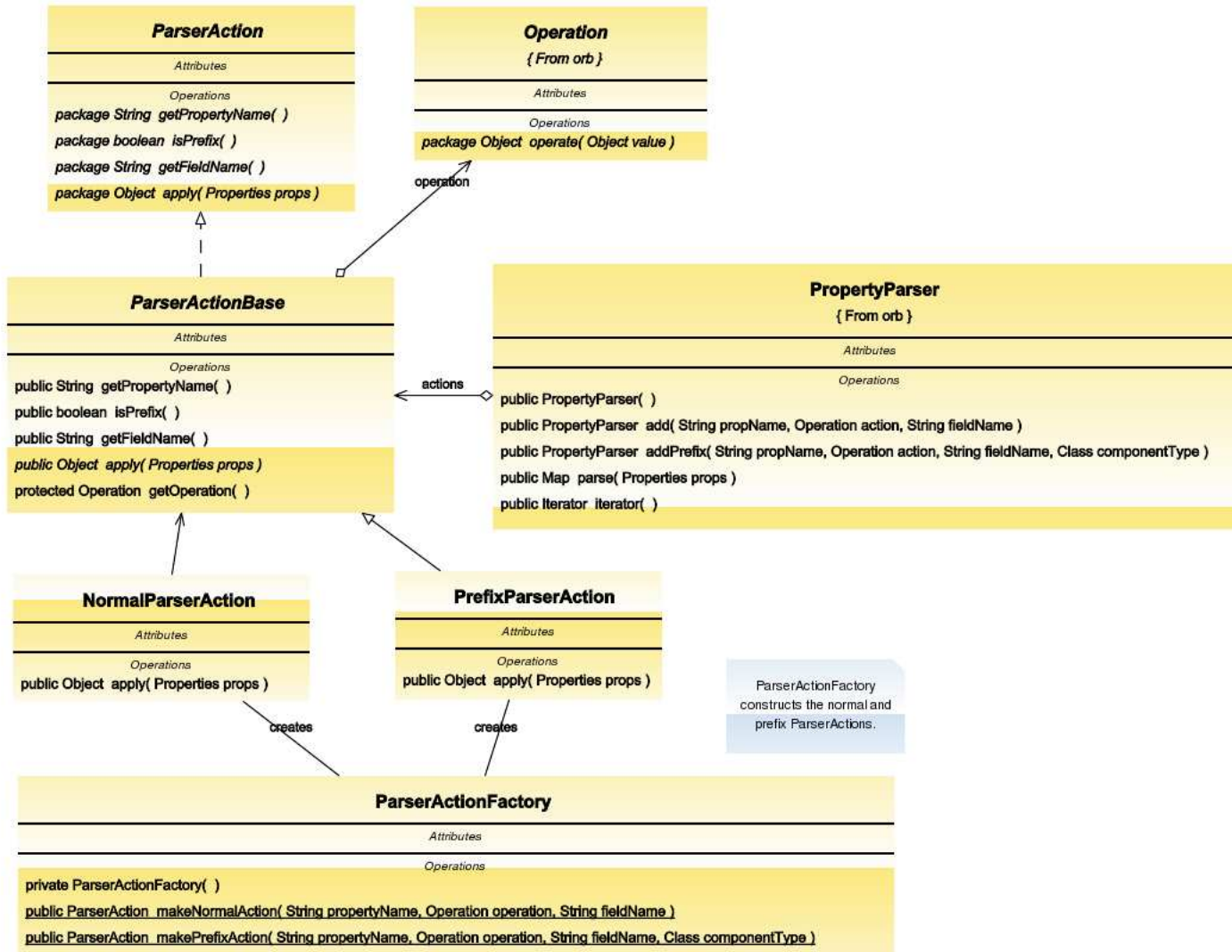
propName which is the property name to which this action is applied

action which is the `Operation` performed by this action

fieldName which is the name in the resulting `Map` in which the result of the `Operation` is stored. This is used later (see 2.3.1.4 on the following page) for storing the results in a configuration object like `ORBData`.

There are two other important methods in the `PropertyParser`. The `parse` method takes a `Properties` instance and returns a `Map` from `fieldNames` to the parsed values. This is the main parsing method in the framework. The `iterator` method returns an `Iterator` over the `ParserActions`, which can be used to find all of the property names in the `PropertyParser`. This is used by the `DataCollector` to determine which properties are required.

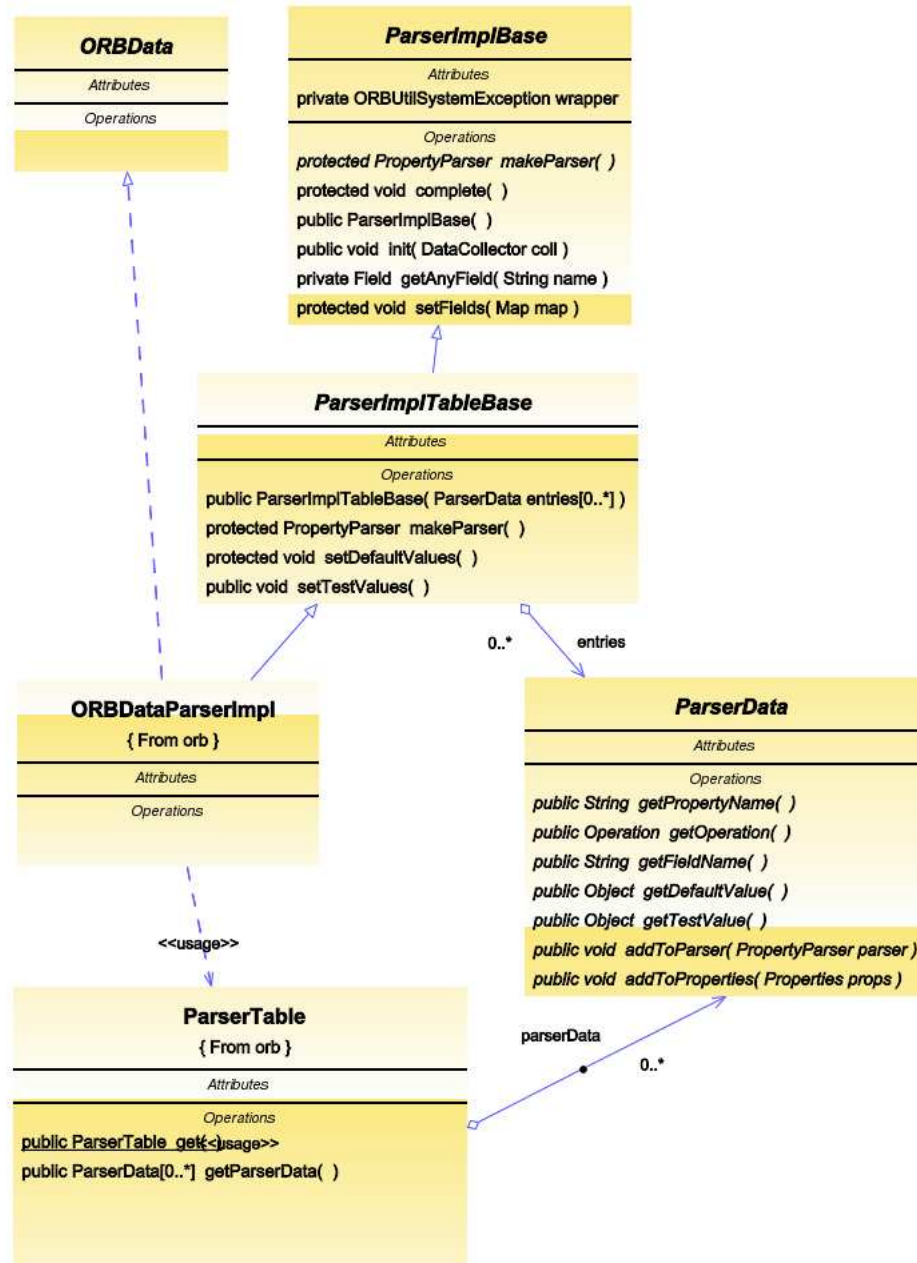
The following diagram shows the different classes used in the `PropertyParser` implementation:



2.3.1.4 Base Classes for Parsing Properties

The top of the configuration framework includes classes that can take the Map returned from a PropertyParser and use it to update the fields in a class that is essentially a read-only JavaBean. This is done in ParserImplBase. A further extension to this class in ParserImplTableBase allows the use of a table of ParserData to initialize the Parser.

The ORB configuration data is represented by the ORBData class. It is implemented by ORBDataParserImpl, which extends ParserImplTableBase and uses ParserTable (essentially a large ParserData[]) to provide the



initialization data.

The ParserData contains the following information:

- The property name.
- The Operation used to parse the information.
- The field name into which the parsed configuration data is placed.
- The default value which is used when the Properties do not contain the property name
- Test data and test value which are used to test the Operation. This is done automatically in the corba.orbconfig test (see testORBData()).

Instances of ParserData are created by the factory methods in ParserDataFactory. ParserData is implemented by the NormalParserData and PrefixParserData classes.

2.3.2 Details of ORB.init

Initializing an ORB from an `ORB.init()` call (the two versions that have arguments) requires several steps:

1. Select the ORB class that needs to be instantiated.
2. Create an instance of the class.
3. Invoke the `set_parameters` method on the instance.

These steps are all standard. The more interesting part is what happens in `set_parameters`.

`set_parameters` proceeds as follows:

1. Call the `preInit` method, which sets up most of the configuration independent parts of the ORB (which is not very much). This includes:
 - (a) Initializing a `PIHandler` that does nothing, so that the ORB can perform requests before PI has been initialized (which happens near the end).
 - (b) Create a `ThreadGroup` for use by the ORB. This is complicated because of some Applet considerations: for details, see the code.
 - (c) Set up the transient server ID. This is currently just set to `System.currentTimeMillis`.
 - (d) Set up the `ORBVersion ThreadLocal`.
 - (e) Initialize some locks.
 - (f) Initialize the various registries.
 - (g) Set up invocation info `ThreadLocal` stacks.
2. Create a `DataCollector` that represents the available configuration data for use in creating this ORB instance. `ORB.init(String args, Properties props)` uses the `NormalDataCollector`, while `ORB.init(Applet app, Properties props)` uses the `AppletDataCollector`.
3. Call the `postInit` method, which handles all configuration-dependent ORB initialization. This includes:
 - (a) Setting up the `ORBData`. This is simple: just construct `configData` using `ORBDataParserImpl` and the `DataCollector`.
 - (b) Set up the debug flags.
 - (c) Initialize the monitoring manager, the transport manager, and the legacy server socket manager.
 - (d) Set up another parser (using the Parser framework) to obtain the `ORBConfigurator`. Run the `ORBConfigurator`.
 - (e) Set up the real `PIHandler`, replacing the no-op version from the beginning of the initialization sequence.
 - (f) Set up the thread pool manager and the byte buffer pool

Most of the detailed ORB initialization happens in the ORB configurator, which we will examine next.

2.3.2.1 The ORB configurator

We have two mechanisms for customizing the ORB initialization: the standard (from PI) `ORBInitializer`, and the `ORBConfigurator`. Why two? There really is only need for one, except for one really irritating problem: the `ORBInitializer` does not provide direct access to the ORB or the ORB configuration data (our `DataCollector`). We also want to be able to have ORB extension parse configuration properties that are not even known in the base ORB configuration (although we don't currently make use of this). So I chose to create the `ORBConfigurator` interface.

Looking back on this now, there is an alternative that may have been better: simply extend `ORBInitInfo` with an internal SPI so that we could access the ORB directly. The current situation is the result of a spec

compromise: no one could agree on what operations should be allowed on an ORB instance while it's in the process of initialization, so a facade object (ORBInitInfo) was specified that sharply restricts what can be done with the underlying ORB instance. Of course, this makes it hard to access anyone's ORB extensions from inside an ORBInitializer.

The current ORBConfigurator we use is replaceable, as is obvious from the use of the parser to obtain it. For example, we could replace the current Java-code driven approach with an XML-based approach, a Lisp-Sexpression approach, something based on the JINI config language (which is an interpreted simple subset of Java), or some other mechanism. But this does not seem to be needed today.

Here is what the ORBConfiguratorImpl configure method does:

1. Initialize the default object copiers. This is overridden in the app server init. Object copiers are discussed in more detail in 9.1 on page 34.
2. Initialize IOR machinery (see 6.2 on page 31 for more details). This involves:
 - (a) Setting up the tagged profile and tagged profile template factories.
 - (b) Registering the tagged component factories. This could be extended by the app server init to include CSiv2 related tagged components, which would remove the need for using the very slow codec APIs.
 - (c) Registering the ValueFactory instances for the ObjectReferenceTemplate (this is needed so that the ORB knows how to marshal these classes, since their public interface is an abstract value type).
 - (d) Register the ObjectKeyFactory.
3. Register the ClientDelegateFactory.
4. Initialize the transport. As noted in the comments, this is complicated because we support several legacy mechanisms for initialization. The more preferred mechanism for intializing the transport is simply to register all required Acceptor instances (but we need a better framework for creating Acceptors easily, I suspect). But we also have the older SocketFactory mechanism, as well as a number of even older configuration parameters. See ?? on page ?? for a discussion about the transport design.
5. Initialize naming. This really means setting up the resolvers for resolve_initial_references and related methods. This provides access to a name service through either the old bootstrap or the standard INS mechanisms. Resolvers are discussed in 8.10 on page 33.
6. Initialize the service context registry. Just as in the IOR case, this could be extended by the app server init to include CSiv2 related service contexts, again avoiding the need for using the codec APIs.
7. Initialize the request dispatcher registry. This is the central mechanism that ties all of the code together that is needed for invoking and dispatching in the ORB. This includes:
 - (a) Registering ClientRequestDispatchers and ServerRequestDispatchers.
 - (b) Registering the special ServerRequestDispatcher used for INS (this one has no object adapter).
 - (c) Registering the LocalClientRequestDispatchers, which are used for co-located requests. This includes all of the servant caching optimizations. One small note on this: we could extend the optimizations significantly to cache the servant in ALL cases, and then have the POA invalidate the cache when necessary. Currently we assume that we are caching only in the ServantLocator case, and we assume that the ServantLocator always returns the same instance for the same object reference. What we have now is fully effective for the App Server, so there has been little incentive to re-visit this issue.
 - (d) register the ServerRequestDispatcher used to handle the bootstrap mechanism.
 - (e) Register the ObjectAdapterFactory.

Much of this registration is driven by subcontract IDs. See 6.1 on page 31 for more details.

8. Register the initial reference for dynamic any support.
9. Handle the psersistent server initialization.

2.3.3 Initializing the ORB in the App Server

TBD

- portable interceptors
- PEORBConfigurator
- The ORBManager

2.4 ORB Shutdown

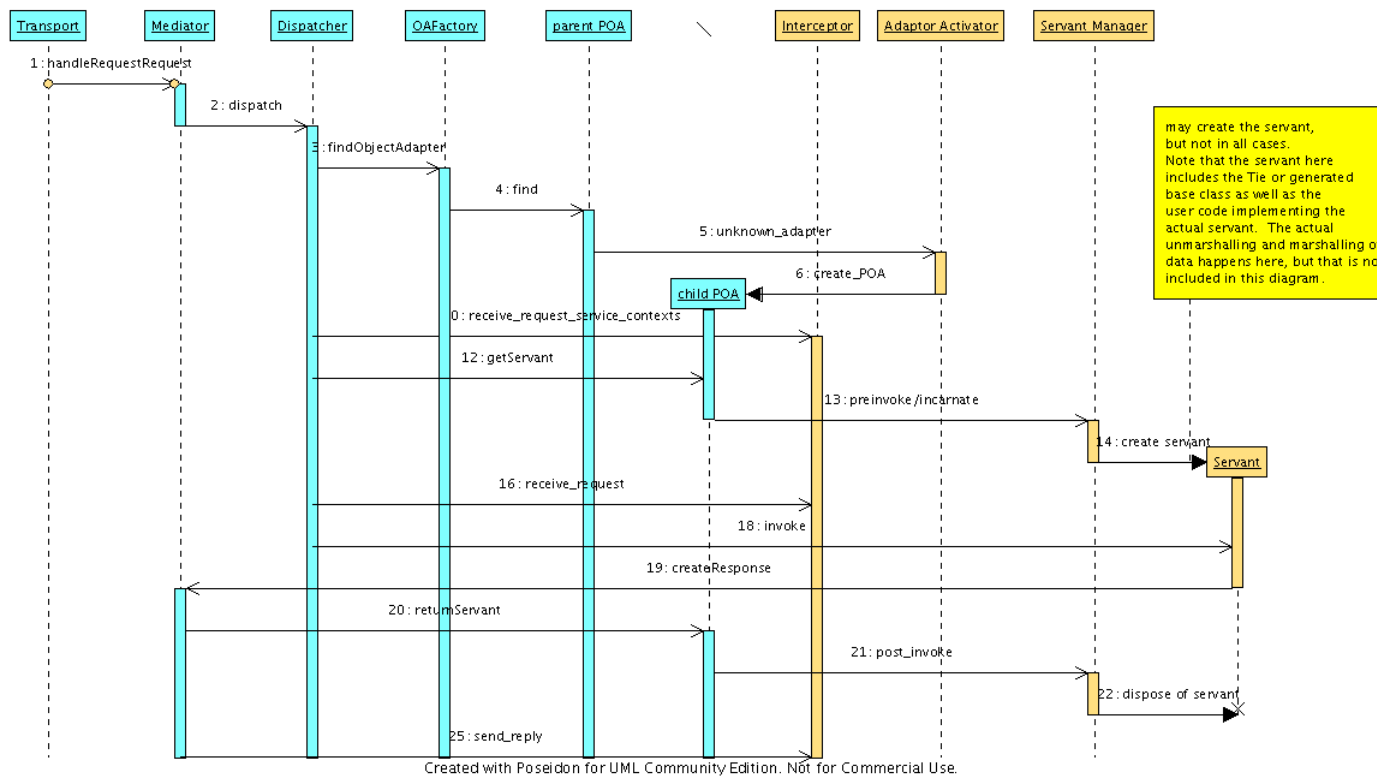
The primary issue in starting up the ORB is simply to configure all of the data needed for running the ORB. In contrast, shutdown must carefully control access to the ORB so that spurious errors do not occur in requests that are in the middle of being processed.

Details TBD.

Chapter 3

Dispatch Path Overview

Here is a simplified sequence diagram describing the overall ORB dispatch path:



this at a high level)

The next few chapters look at this in more detail through the PEPt model.

Chapter 4

Presentation

4.1 Stubs and Skeletons

See mapping specs and dynamic RMI-IIOP document

4.2 Data types

IDL: basics

IDL: typecode and any

Java additions and value types

Chapter 5

Encoding

impl.encoding
impl.io
impl.util

5.1 Repository IDs

Chapter 6

Protocol

impl.protocol
impl.protocol.giopmsgheaders

6.1 Subcontract IDs

6.2 IORs

6.3 Service Contexts

6.4 GIOP Message Representation

Chapter 7

Transport

The transport is responsible for handling the transfer of data to and from endpoints. Connection management is also an important part of the transport, since we most commonly use GIOP as a protocol, and GIOP is connection based. We also include here the logic that is used to decide which of several possible endpoints should be used for a connection.

The client and the server roles in a CORBA request are distinct, but both are event driven: messages are normally received by a selector thread. The client simply needs to get a connection, write the messages to the connection, and wait for a response. The server is event driven: it responds to messages received. The server also contains acceptors, which represent endpoints on which the server listens for new connections,

- connection management
- mapping endpoints to sockets
- message tracing
- acceptors
- listener
- selector

Chapter 8

Other Aspects of the ORB

- 8.1 Object Adapters
- 8.2 The RequestDispatcherRegistry
- 8.3 Encoding Details
- 8.4 ORB Logging
- 8.5 ORB Monitoring
- 8.6 ORB versioning
- 8.7 ORBD and Server Activation
 - 8.7.1 current model
 - 8.7.2 ideas for using ORT
- 8.8 Portable Interceptors
- 8.9 RMI-IIOP Implementation
- 8.10 Resolvers
- 8.11 Name Services
- 8.12 ORB and App Server Integration

Chapter 9

Utilities

- 9.1 Fast Object Copying
- 9.2 Dynamic Code Generation
- 9.3 Useful utilities
- 9.4 FSM Framework
- 9.5 Graph Utilities
- 9.6 JDK 5 Specific Utilities
- 9.7 Timing Framework

Chapter 10

Living with our legacy

10.1 Testing Principles

10.2 Benchmarking

10.3 FOLB Support

10.4 HWLB Support

Chapter 11

Compilers

11.1 New rmic iiop backend

11.2 idlj

Chapter 12

Future Directions

12.1 Embedded Languages

12.2 Components

12.3 Fast Marshalling

12.4 Security

include security document here (that I was working on for a while last summer)

12.5 Better handling of Invocation Info