CPP

Table of contents

1. Introduction

This section describes the C++ implementation of XLattice. Although this set of documents -- the XLattice Web site -- describes cpp as a component of XLattice, and though the source code is organized as though it were, in fact cpp contains all of the XLattice components. For each there is a cpp library and a corresponding include file.

2. Proposed Approach [as of 2006-04-25]

The C++ code is in the cpp component and so found under $XLATTICE_HOME/cpp. This more or less mirrors the structure of the Java code. The major difference is that much of the functionity now in the Java xlattice/util component is in xlattice/cpp/core. However, some of this may wind up in xlattice/cpp/util.

The reason for the change in approach is that in the Java code the basic XLattice abstractions are collected in the util component's src/java/org/xlattice directory, with actual utility code one directory below that, in src/java/org/xlattice/util and subdirectories. This is a bit confusing. In all other cases, all non-test code for the component is under src/java/org/xlattice/$COMPONENT_NAME

2.1. Directory Structure

There are 6 [8] component directories below xlattice/cpp - core, [util, corexml,] crypto, transport, protocol, overlay, node, where those in square brackets are currently absent. In addition there are include/ and lib/ direcories at the same level, and a perl/ directory, about which more later.

Each component directory has src/, test/, and build/ subdirectories. These may have sub-subdirectories. There are C++ source files in src/ and test/. These are compiled to build/. Compiled code from src/ goes into the library for the component and any deliverable executables. Compiled code and executables from test/ goes into build/ but not into the component library.

In each component directory there is a Makefile. This is generated by a Perl script, perl/makemake.pl. These are not recursive Makefiles. Each includes body.mk from the topmost component directory. Each component is assumed to depend upon the library from the next-lower component. Transport, for example, depends upon crypto which in turn depends upon core. Therefore each body.mk includes body.mk from the earlier component. The result is that invoking make in cpp/transport will result in as many lower-level libraries being rebuilt as necessary, but does not affect anything in higher level components.

This is not yet implemented, but the intention is that the Makefile should automatically run unit tests and only build the component library if the tests succeed. That is, the Makefile

should have much the same targets as XLattice's Ant build.xml.

cppunit doesn't seem appropriate, so the plan is to do a compatible xunit implementation, tentatively called cpptest.

2.2. Functionality

This is largely a discussion of what goes into the cpp/core component.

As I see it now, at least the following should be part of cpp/core:

- cpptest, the unit test package
- a simple garbage collection scheme built around reference-counting objects
- immutable Strings, as familiar from Java; these may not be guaranteed to be unique
- StringBuffers, as from Java
- callbacks, as in the CryptoServer
- an events package

The objective is to be able to build single-threaded XLattice nodes. In order to be single-threaded, these have to be organized around events: when code would otherwise block, it registers a callback to handle the pending event. Whenever it would go idle, it checks for ready events and handles them using the registered callbacks. Because in an event-oriented environment it is difficult to track when objects can be disposed, we need some sort of automatic garbage collection. Necessarily this involves reference counting. Many, possibly most, of the objects that need reference counting will be Strings. Experience with Java shows that where you have immutable Strings subject to GC, you also need StringBuffers, to efficiently build the Strings and to avoid polluting the world with tiny little immutable Strings, fragments of Strings under construction. Finally, all of this is going to be very complicated and error-prone, so we need the basic tool required for test-driven development: an xunit implementation suitable for our environment, cpptest.

Also part of cpp/core will be basic XLattice abstractions: Acceptor, Connector, Connection, and Transport; DigSigner, Key, PublicKey, Secret, and SigVerifier; Node and Peer. It is very likely that the crypto stuff will just be a wrapper around crypto packages, most obviously OpenSSL. Tentatively we will try to make this look like Sun's JCE, where different providers can be plugged into a common interface.