# Peer to Peer and Mobile Systems Support

Semester Thesis

Rowan Klöti

19th July 2011

**Advisors**: **Remo Meier**
**Supervisor**: **Prof. Dr. Roger Wattenhofer**
Computer Engineering and Networks Laboratory, ETH Zurich

**Abstract**

In this work, the *Sharpen* Eclipse plugin, created to convert Java to C# code, is adapted and modified to output code in C++ instead. Furthermore, certain parts of the Java API are adapted to be used by converted code. The intent is to allow the *Pulsar* content distribution system to be ported to platforms where no Java or C# environment is supported.

# Contents


**1 Introduction** **2**

**2 Related Work** **3**
2.1 Brief overview of Sharpen . . . . . 3
2.2 Other software frameworks used . . . . . 4
2.2.1 Java Native Interface (JNI) . . . . . 4
2.2.2 PThreads . . . . . 4
2.2.3 C++ Standard library classes . . . . . 4

**3 Sharpen for C++** **5**
3.1 Overview of architectural changes . . . . . 5
3.2 Changes in detail . . . . . 7
3.2.1 Basic syntax . . . . . 7
3.2.2 Strings . . . . . 8
3.2.3 Arrays . . . . . 8
3.2.4 Enumerations . . . . . 9
3.2.5 Classes and interfaces . . . . . 10
3.2.6 Namespaces and compilation units . . . . . 11
3.3 The CS/CPP AST . . . . . 12

**4 Java API** **17**
4.1 Networking . . . . . 17
4.1.1 Address resolution . . . . . 17
4.1.2 Datagram sockets . . . . . 18
4.2 Threads . . . . . 19
4.2.1 Creating new threads . . . . . 19
4.2.2 Mutexes and condition variables . . . . . 19

**5 Future Work** **20**
5.1 Synchronisation . . . . . 20
5.2 Collections . . . . . 21
5.3 Exceptions . . . . . 21
5.4 C++ 2011 . . . . . 22

**6 Conclusion** **23**

# Chapter 1

# Introduction

Pulsar, also known as StreamForge[1], is a distributed peer-to-peer system which allows media to be streamed between clients. It has been offered as a commercial product by a spin-off venture since 2009. It is desired to port the client solution to as many platforms as possible. The client itself was developed on the Java platform. However, many potential platforms, especially mobile platforms, do not support the JVM. For this reason, the Computer Engineering and Networks Laboratory has made use of the *Sharpen* Eclipse plugin, originally developed by *Versant* to automatically translate their object oriented database *db4o* into C#[2], allowing the client to be used on .NET platforms. In this work, the existing plugin will be adapted to output C++ rather than C# code, to further extend the number of platforms that the Pulsar client can be easily ported to.

The primary aim of this semester thesis is to transform syntactically valid Java code into functionally equivalent syntactically valid C++ code. In order to actually translate Java to C++, this is not sufficient. In particular, it is necessary to translate parts of the Java API to C++ in order that they can be called from the converted code. *Sharpen* has a mechanism to map namespaces, classes, properties and methods, but this is not sufficient as the Java standard API has a much greater scope than C++ standard library and many of the classes are not entirely semantically isomorphic. This project has not attempted to recreate the Java API in C++, nor to find a C++ equivalent for every class in the Java API. Indeed, the developers of *Sharpen* make no claim on the completeness of the conversion: They clearly state that it is necessary to write code that is compatible with it.

[1] See `http://www.streamforge.com/`

[2] See `http://developer.db4o.com/Projects/html/projectspaces/db4o_product_design/sharpen.html`

# Chapter 2

# Related Work

## 2.1 Brief overview of Sharpen

Sharpen[1]is an Eclipse plugin which consists of several important components.

- The Eclipse framework itself, which provides an AST representation of Java syntax. This is required in order for Eclipse to parse and modify the Java source code. This is required for the conversion.
- The CSharpBuilder class. Broadly stated, it obtains an Eclipse output tree, parses it with a visitor[2] pattern, then outputs an C# AST. There are a number of derived classes performing specialised functions, such as AbstractNestedClassBuilder, which is for parsing nested classes (classes defined inside another class).
- The C# AST elements. Unlike the Java AST, this is *not* provided by the Eclipse environment. Instead, it is part of the *Sharpen* project.
- The CSharpPrinter class. Also implementing the visitor pattern, it parses the C# AST and outputs syntactically valid C# code.
- The Configuration class. Amongst other things, it configures mappings between Java and C# classes, methods and properties and configures various parameters of conversion.
- A variety of internal classes, which will be discussed in more detail as needed later on. They include NamingStrategy and Mappings.

[1]The original codebase of Sharpen can be viewed at `http://source.db4o.com/db4o/trunk/sharpen/sharpen.core/src/sharpen/core/`. Please note that it does *not* correspond with code used here, even without the modifications made to support C++.

[2]See `http://en.wikipedia.org/wiki/Visitor_pattern` for more information about this pattern.

## 2.2 Other software frameworks used

### 2.2.1 Java Native Interface (JNI)

Java Native Interface (JNI)[3] is an adaptation layer allowing Java classes to use native C or C++ code. For any method declared with the NATIVE keyword, the necessary declaration is generated automatically. This was used to implement networking and threading routines in C++ that could be called from Java. These would then be adapted to run C++ code directly. There are several caveats when using JNI, most originating in the JVM's memory management and garbage collection routines.

### 2.2.2 PThreads

PThreads (short for POSIX threads)[4] is the standard thread library for POSIX systems, such as Linux. Although all implementation here was done on Windows, PThreads is still preferred to a native Windows solution, as PThreads offers the only somewhat platform-independent solution for threads available at the current time. PThreads was used to implement the creation of threads, as well as methods to lock and unlock mutexes, to wait and to notify threads.

### 2.2.3 C++ Standard library classes

The C++ standard provides for a variety of classes, of which only two are used in this project:

- The STD::STRING class[5]. This is the preferred alternative to CStrings (arrays of CHARs) in C++. They are mutable, supporting common operations such as concatenation (by overriding the "+" operator), assignment or substring finding. The class manages memory automatically, allocating new space as needed. They are used in lieu of JAVA.LANG.STRING class, so mutable strings are not needed. Rather, they are used like Java strings, with new ones being created whenever a string operation (e.g. concatenation) is performed.

- The STD::VECTOR class[6]. This is one of various generic containers supplied by the STL (standard template library). It is intended as an alternative to arrays and is used to replace Java arrays. Like the STD::STRING it can grow and shrink as needed, reallocating memory and copying its contents if needed and it guarantees contiguous memory assignment. Also like strings, not all of the functionality provided by the class is required. They support access via the index operator or with the AT() method, the latter throwing an exception if the index is out of bounds.

[3] See online manual at `http://java.sun.com/docs/books/jni/`
[4] Online tutorial at `https://computing.llnl.gov/tutorials/pthreads/`
[5] Online reference: `http://www.cplusplus.com/reference/string/string/`
[6] Online reference: `http://www.cplusplus.com/reference/stl/vector/`

# Chapter 3

# Sharpen for C++

## 3.1 Overview of architectural changes

Here is a brief summarisation of the main classes that I have added to *Sharpen* in order to accommodate Java-to-C++ conversions. Details of how different parts of the language were adapted are discussed in the next section.

- CPPBuilder: In general, I have attempted to adapt *Sharpen* with as few changes to the CSharpBuilder class as possible. There are several child classes of CSharpBuilder, so this must be taken into account when making modifications. I have chosen to add new class, CPPBuilder, which derives from CSharpBuilder. This class only overrides methods when it is neither possible to make the changes further along the conversion stack, nor to make changes which are language-agnostic, for instance by the use of factory classes.

- CPPSourcePrinter and CPPHeaderPrinter: These classes replace the CSharpPrinter (they do not derive from it, but rather from its parent class CSVisitor). For each class, a class declaration is produced in the header file, and a class definition is produced in the source file. It is here that the majority of adaptations were made.

- AbstractASTElementFactory: Where it was necessary to replace CS AST element classes, the CPP equivalent classes derive from the originals. The AbstractASTElementFactory class implements abstract factory pattern[1] and has two concrete implementations, CSASTElementFactory and CPPASTElementFactory. The factory methods pass on all required information needed to construct the CPP objects, the extra information is simple ignored by the CS class. As all of the CPP elements derive from equivalent CS methods, it would have been possible to replace

[1] See `http://en.wikipedia.org/wiki/Abstract_factory_pattern` for more information about this pattern.

them entirely or even simply modify the base class, but I have attempted to limit changes to the parts of the code relevant to CS conversions to avoid introducing bugs there. The classes that I have overridden are:

  - CSReferenceExpression to CPPReferenceExpression
  - CSMemberReferenceExpression to CPPMemberReferenceExpression
  - CSTypeReference to CPPTypeReference
  - CSCompilationUnit to CPPCompilationUnit
  - CSParenthesizedExpression to CPPDereferenceExpression

- CPPConfiguration: This class derives from the Configuration class. It provides mappings between Java and CPP methods, although only a few of these are implemented. Furthermore, in conjunction with the CPPNamingStrategy and CPPMappings, it maps fully qualified type names[2].

- CPPNamingStrategy and CPPMappings: As mentioned above, these classes are responsible for mapping type names. The main changes required here are to accommodate the differences between C# type names (which are essentially equivalent to Java type names) and C++ type names, where it is necessary to distinguish between namespace part and the class name part (which may consist of several nested classes). The mappings class is an implementation of the Mapping interface. There are several other services provided as an implementation of a interface, although this was the only one that needed to be overridden.

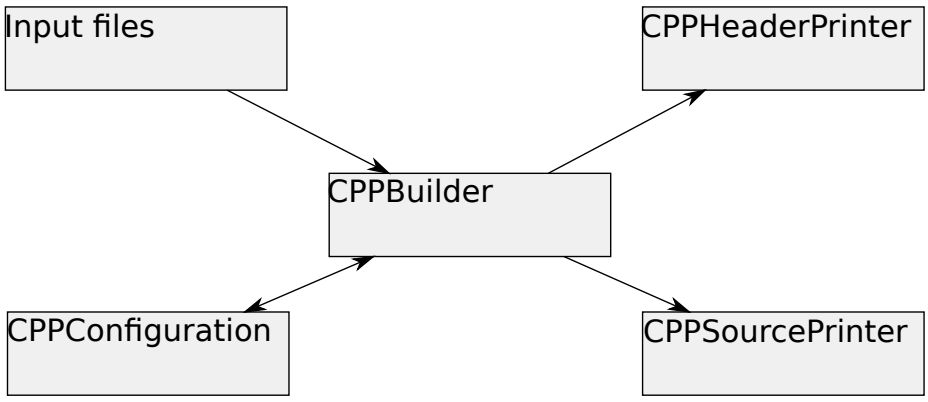


Figure 3.1: Simplified diagram of revised *Sharpen* structure.

[2]Fully qualified means that the name is supplied with the package name (in Java) or the the namespace name (in C# or C++).

## 3.2 Changes in detail

### 3.2.1 Basic syntax

Fortunately, Java, C# and C++ share many elements of their basic syntax (they are so-called *"curly bracket languages"*). IF/ELSE, FOR, WHILE, DO/WHILE, SWITCH/CASE and TRY/CATCH blocks are translated verbatim. Likewise, the RETURN, NEW, THROW, CONTINUE and BREAK statements are translated without any changes. The DELETE operator does not occur in Java and is never generated by *Sharpen* - Garbage collection must be used if any objects are generated on the heap. In general, each Java statement corresponds with a C++ statement and each Java code block corresponds with a C++ code block. Variable declaration is considerably different: All Java objects are either a primitive[3] or an object[4], including enumerators, strings and arrays, which are special cases and are dealt with below.

All Java primitives correspond to a C++ primitive and are translated as such. Java objects cannot be translated directly, however. Java objects are addressed with references, which, in spite of the name, actually correspond more closely to C++ pointers than C++ references. In particular, it is impossible to point a C++ reference at a new address once it has been initialised, and all C++ references can only be initialised at declaration, not later on. For this reason, I have chosen to map Java object references to C++ pointers. The CSTypeReference AST element, which is used to represent variables' types, does not carry any information on whether or not the type being referenced is a primitive or not. Although it would be possible to directly compare the typename, this solution would be inelegant and is not future-safe, as new primitives could be added in the future[5]. Instead, I have added a new AST element, CPPTypeReference, which contains information about whether or not it is a primitive. The header and source printers then add a dereferencing operator as needed, such as in variable declarations. Type casting is translated as C style type casts - that is, (*typename*) *expression* - for primitives and as DYNAMIC_CAST<*typename*> for non-primitives (It is not possible to use DYNAMIC_CAST for primitives as it can only be used to convert pointers and references).

A further difference in basic syntax is the member reference operator. In the Java the member reference operator "." is used for all member references and corresponds to both the member reference operator and scope resolution operator in C++. Apart from the scope resolution operator, it is necessary to distinguish between access to static and non-static fields as well invocation of static and non-static methods. For this reason, I have created the class CPPMemberReferenceExpression, derived from CSMemberReferenceExpression, which also contains information about static/non-static access. Static access is done via the scope resolution operator, while non-static access is done with the

[3] INT, SHORT, LONG, BOOLEAN, CHAR, BYTE, FLOAT or DOUBLE

[4] That is, a subclass of JAVA.LANG.OBJECT

[5] Consider the LONG LONG primitive, for an integer of at least 64 bits, added to C and C++.

arrow expression "->", as all classes are accessed via pointers. Objects are never allocated on the stack, reflecting the lack of support for stack-based storage of objects in Java.

### 3.2.2 Strings

Both Java and C++ provide a string object. CPPSharpen uses a pointer to a STD::STRING object whenever it encounters a JAVA.LANG.STRING. In particular, a C++ string literal, which returns a CString, must be enclosed by a STD::STRING constructor invocation, using the CSCONSTRUCTORINVOCATIONEXPRESSION element. This may result in the unnecessary instantiation of objects, and underscores the necessity of using a garbage collector in conjunction with a converted program, as neither Java not C# have a DELETE operator or any equivalent alternative.

It is also necessary to handle string concatenation separately, as Java allows concatenation with the "+" operator, a deviation from the general principle of not allowing operator overloading in Java. Although STD::STRINGs support concatenation with the "+" operator, like all of the STL classes, they require references instead of pointers. To accommodate this, the visitor for the INFIXEXPRESSION node has been overridden. The new handler is used only when either side of an INFIXEXPRESSION has a STRING binding. Then the left hand and right hand expressions are converted to CSEXPRESSIONS with the MAPEXPRESSION method. Both expressions are embedded in a CPPDEREFERENCEEXPRESSION, then added to a new CSINFIXEXPRESSION and this finally to a CSCONSTRUCTORINVOCATIONEXPRESSION. If there are more than two operands, the extra ones are contained in a list. These are then processed as above, with the entire previous expression as the left side of a new CSINFIXEXPRESSION. The net effect is that in each infix operator, each of the operands is dereferenced to a STD::STRING, the standard concatenation via OPERATOR+ is performed, then a new string with the result is constructed. This is repeated recursively if there are more than two operands.

In order to handle STD::STRING methods such as COMPARE, it is necessary to first dereference the STD::STRING pointer. This is achieved by overriding the visitor for METHODINVOCATATION AST nodes. A new method, ISNATIVEMETHOD, identifies calls to standard library functions (currently only STD::STRING methods) and a second method ISNATIVEARGUMENT identifies all arguments which require dereferencing. These are then stored in a CPPDEREFERENCEEXPRESSION. This could be expanded in future to allow more native methods to be used.

### 3.2.3 Arrays

Java arrays are objects which support bounds checking and "know" their maximum size. Standard C++ arrays, either declared statically or with the NEW operator, do not do bounds checking and do not record their size, at least not in a manner available to the programmer. It is therefore necessary to provide a

replacement. I have elected to use the STD::VECTOR container, which actually supports automatic resizing (which is not necessary, as Java arrays cannot be resized) and bounds checking. Once initialised, it provides a reference to a its contents via an index expression, allowing it to be used just like an array.

Unfortunately, it is not possible to instantiate a STD::VECTOR with an array literal. It is therefore necessary to create a temporary constant local variable, named __INIT_*arrayname*, where *arrayname* is the name of the array to be declared. The array is then initialised with the iteration constructor, which takes the start and end[6] points of the array. If no initialiser is supplied, only the size of the array, then the array is created with the repetitive sequence constructor, using the default constructor of the element type.

The LENGTH property of the Java array is mapped to the LENGTH() method of STD::STRING. Currently, the use of iterators is not supported. Java and C++ iterators have very different semantics, with Java iterators using the NEXT() method and C++ iterators supplying the BEGIN() and END() methods and overloading the incrementation operator. Since C++ currently has no syntax for a FOREACH block, unlike Java and C#, one typical usage of iterators would not be possible to implement.

### 3.2.4 Enumerations

As with arrays, Java enumerations, while declared in the same way as C++ enumerations, are considered to be objects by the language - a subclass of JAVA.LANG.ENUMERATION - while instantiated C++ enumerations are considered primitives. This implies that Java enumeration can and do have methods, which C++ enumerations can and do not. This problem is also present for C#, in which enumeration are also primitives. The solution chosen by the Computer Engineering and Networks Laboratory was to create a supplemental "extension" class which would implement the missing methods statically. This solution was maintained here, was changes being made as necessary to make the generated code C++ compliant. The extensions classes generated supply a static array VALUES containing all possible values of the enumeration as well as the following static methods:

- VALUES() - Returns the array VALUES
- VALUEOF(STRING) - Returns the enumeration value with the name specified in the parameter
- NAME(ENUM) - Returns the name of the enumeration value specified in the parameter

In addition, an extra value _NULL is added to the set of enumerator values. As Java enumerations are objects, accessed via references, it is necessary to somehow represent the value of a null reference in C++ or C#.

[6] Actually one step beyond the end of the array

### 3.2.5 Classes and interfaces

This is where most of the important changes were made. Where Java and C# classes are declared and defined in the same place, C++ classes are typically declared in a separate header and then defined in a source files. This architecture has been used here - The CPPHeaderPrinter class and the CPPSourcePrinter class, both derived from the CSVisitor class, are executed in serial, producing a header file and a source file for every Java class encountered. Classes defined within another class do *not* produce separate files. Interfaces are treated as classes, as C++ does not support a special INTERFACE type. The classes produced from interfaces contain only public purely virtual methods[7].

The class declarations are produced analog to the original Java classes, with several modifications. The header printer collects all the members and sorts them according to visibility, printing their declarations and the appropriate header. The class methods do not contain a body, the body is defined in the source file instead. Otherwise, the methods are printed with mapped modifiers, parameters and return types. All non-static methods apart from the constructors and destructor are declared as virtual, even if they are not overridden in derived classes. Abstract methods become purely virtual methods, with "= 0" added to the definition. Static methods become C++ static methods. The constructors are printed as-is, the destructor is printed only if the class overrides the FINALIZE method, with this method then becoming the destructor.

Constant fields are printed verbatim. Non-constant fields have their initialiser removed, as C++ does not allow non-constant fields to be initialised in the class declaration. Instead, static fields have their initialiser moved to the source file, while non-static fields have theirs moved to each defined constructor, with a new default constructor being created if none existed beforehand.

Types, that is, classes, interfaces and enums, are declared recursively in the same manner as the containing class: Without definitions. All anonymous classes are translated to a nested class with a random name, as C++ does not support anonymous classes. The extension classes referred to in 3.2.4 on the preceding page are created *outside* the class in which the enumeration is declared, at the same level as the declaring class. If extension classes are used, the main class[8], is printed first. This is done for the sake of extension classes: The STD::VECTOR class requires any type used to be fully defined and not just forward declared, so that it can allocate the correct amount of memory per element. Since the enumeration declaration is in the main class, the main class must be declared before the extension classes.

The source printer prints out all methods declared in *any* class in the header file, and also any static fields. There is no indentation and everything is declared "naked" in the source file without any surrounding blocks. Non-static fields and type declarations are not printed. Also, any method declared as NATIVE is not

[7] A purely virtual method, that is, a method with no definition in this class, corresponds to an abstract method in Java and C#. Any class which contains such a method is an abstract class and cannot be instantiated.

[8] The class with the same name as the compilation unit

printed here, as such methods would simply be printed without a body. For each member defined, the full list of containing classes, usually one but possibly more if nested classes are used, are printed out with the scope resolution operator. In order to keep track of this, the source printer keeps a list of all current classes in a stack. Method bodies are reproduced verbatim, except in the case of constructors, where there may be statements added to define non-static non-constant fields, potentially with an extra line for initialisation if the variable is an array.

Java classes can only inherit from a base class, although they can implement an unlimited number of interfaces. The C++ classes generated inherit the base class first and then all classes created from interfaces - there is no difference between the actual base class and the interface classes in C++. All inheritance is PUBLIC, as this corresponds semantically with Java inheritance and Java does not allow any other form of inheritance. The base class for the current type is stored in a stack. If base constructors are called by the class, the base class constructor name is added to constructor definition header of the derived class - C++ does not have a BASE or a SUPER keyword and does not allow chained constructor invocations to be defined in the class declaration.

If the main class contains a STATIC VOID MAIN(STRING[]) method, then an addition INT MAIN(INT ARGC, CHAR **ARGV) function printed outside that class. This is the *only* time that a function (as opposed to a method) is generated. The function takes that input arguments, initialises a STD::STRING for each one, then writes the STD::STRINGs into an STD::VECTOR and calls the MAIN method with that vector. This allows the translated Java code to access arguments in the normal manner.

### 3.2.6 Namespaces and compilation units

All Java classes are contained in a package, declared with the PACKAGE statement at the start of the compilation unit. The nomenclature for packages generally reflects the directory structure, although the package namespace is flat, i.e. it is *not* possible to embed one package inside another. The use of names divided by a point merely suggests a hierarchical structure. All Java type names can be referred to as a so-called "fully qualified name", which includes the full package name, then the type name. It is also possible to omit the package name, if the IMPORT statement is used. Packages provide an extra layer of encapsulation, preventing name conflicts.

In order to obtain the same effect in C++ (or C#), the use of namespaces is required. Namespaces are declared in a block with the NAMESPACE keyword. It is possible to nest namespaces inside each other. Following the Java convention, no use is made of this capability, all namespaces are flat. As with Java, it is possible to fully qualify a type name with a namespace, using the scope resolution operator "::". Alternatively, this may be avoiding, either by importing a single type into the current namespace, with the USING keyword, or by importing an entire namespace, with the USING NAMESPACE keyword.

*Sharpen* keeps track of all namespaces used in a class, importing all of the

namespaces so that fully qualified namespaces are generally not needed. All namespaces are mapped internally by CPPConfiguration, so if an equivalent C++ class is supplied for a Java class, the namespace can be transformed as well as the class name itself. Each package becomes its own namespace, with the "." separator being replaced with an underscore, as C++ does not allow points in namespace names. The scope dereferencing operator is only used at the end of the namespace, to separate the namespace from the class name and any internal class names. In order to differentiate between the namespace and class part, the CPPMappings class uses the type binding to find out the package name, then cut the package name from the fully qualified name, leaving the class names.

In addition to namespaces, C++ also requires the use of #include directives. The source file automatically includes the header file for the same class. To implement this, the CSCompilationUnit class is overridden by CPPCompilationUnit class, the latter including the property fileName. This stores the file name of the compilation unit. The header file contains an #include directive for each header file of a class referenced by the class under consideration. The CPPConfiguration class, when it performs the type name mapping, also registers the classes referenced in a set, which is also included in the CPPCompilationUnit. Each of #include directives produced has a relative path name. The assumption is that the original code is in a directory structure which reflects the package names. To find the correct path, the CPPConfiguration class counts the number of name space parts (separated by an underscore) in the current namespace and invokes the ".." directory the same number of times. In addition, there are a number of includes in every header, namely for <vector> and <string>. Although it would have been possible to only include these if needed, the extra code complexity would hardly have merited the small savings in compile time.

## 3.3 The CS/CPP AST

In this section, I will provide a brief overview of the AST used to generate C# or C++ code. This is both to aid understanding of the conversion and show the internal representation of the code, as well as an aid to anyone who wishes to further develop the plugin, as there is no extant documentation that I am aware of. I will not provide any information about the Java AST, as this is adequately documented online[9]. Not all possible elements are listed, some are not generally produced by Sharpen, others are not interesting (e.g. documentation and comments). All of the following classes are derived from CSNode, the base class of all AST elements:

- CSCaseClause - Has a list of expressions and a body of code. When multiple expressions are used, the individual cases are written sequentially, allowing multiple expressions for one block of code.

[9]See `http://help.eclipse.org/galileo/nftopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html`

- CSCatchClause - Has an exception (as a variable declaration) and a body of code. Can also be anonymous, specified with boolean variable.
- CSCompilationUnit and CPPCompilationUnit - Representing an individual file of code. Has a list of types (generally with just one class), a namespace, a list of usings and a list of comments. CPPCompilationUnit adds a file name (so that the source file includes the right header) and a list of files to include.
- CSEnumValue - Has a string for a name.
- CSExpression - base class for all expressions, that is, a piece of code that represents some kind of value.
  - CSAbstractInvocation - The base class for all method invocations. Has a list of argument expressions.
    - CSMethodInvocationExpression - This class represents all method invocations. Adds an invoking expression (generally a reference expression) and type arguments, for use with templates.
      - CSConstructorInvocationExpression - This represents the invocation of a constructor with the NEW keyword. Does not add any new members.
  - CSArrayCreationExpression and CSArrayInitializerExpression - They are used for array initialisations. CSArrayCreationExpression can either represent a new empty array with a fixed length (represented as an expression) and an element type, or the creation of an array literal with an element type and a CSArrayInitializerExpression, which contains a list of expressions to be printed out between curly brackets.
  - CSBaseExpression - Represents invocation of base class constructor. Special work around needed, as C++ does not have a BASE keyword. See 3.2.5 on page 10.
  - CSBoolLiteralExpression, CSCharLiteralExpression, CSNumberLiteralExpression, CSStringLiteralExpression - Represent all kinds of literals. Contains the value of the literal to be included. The string literal also provides an escaped value, which includes the quotation marks around the string.
  - CSNullLiteralExpression - Used for null pointer. No members. In C++, prints out a "0".
  - CSCastExpression - Has an expression to cast and a type reference to cast it to.
  - CSConditionalExpression - Used for the ternary operator "*condition_expression* ? *expression_if_true* : *expression_if_false*". Contains all three expressions as members.

- CSDeclarationExpression - Contains a list of variable declarations.
- CSIndexedExpression - Used for index access to an array or container. Translated with "->operator[]()". Contains expression (usually a reference) and a list of expressions for the indexes
- CSInfixExpression - Contains two expressions (left and right) and a string and represents expressions of the form "*left OP right*". E.g. $a + b$
- CSParenthesizedExpression - Contains an expression and represents expression surrounded by parentheses.
  - CPPDereferenceExpression - As above, but add dereferencing operator *"*"* before parentheses.
- CSReferenceExpression and CPPReferenceExpression - Has a string and generally represents variable or method names. CPPReferenceExpression also has a boolean to mark whether the reference is to a primitive. In a declaration, non-primitives need a referencing operator so that they are declared as pointers.
  - CSMemberReferenceExpression and CPPMemberReferenceExpression - Contain an expression and a string and represent expressions of the form *expression.string*, such as used in method invocation or field access. CPPMemberReferenceExpression also has a boolean for static access, in which case the scope resolution operator *"::"* is generated.
- CSThisExpression - Represents the this keyword. No members.
- CSTypeReferenceExpression - Base type for all type names, that is, anything that can be declared as a variable.
  - CSArrayTypeReference - Represents array type. Has an integer for the number of dimensions and a further CSTypeReferenceExpression for the element type.
  - CSTypeReference and CPPTypeReference - Represents a normal type. Has a string for a type name and list of CSTypeReferenceExpressions for type arguments. The latter are used in case the type is a template: They are printed between the angular brackets.
- CSUnaryExpression - The base class for all expressions with just one operand. Has a string for the operator and an expression for the operand.
  - CSPostfixExpression - Expressions of the form *OperandOP*. E.g. $i{+}{+}$
  - CSPrefixExpression - Expressions of the form *OPOperand*. E.g. *!a*

- CSMember - base class for all members of classes and interfaces. Has a string for the name and for the signature and a CSVisibility object for visibility[10].
  - CSMethodBase - The base class for all declarations of methods, anything that can be called. Has a list of variable declarations for parameters and a body. There is a boolean for variadic arguments, but this is not currently supported for C++.
    * CSConstructor - Has a CSConstructorModifier for a modifier[11] and a CSConstructorInvocationExpression for chained constructor invocation (to call base class constructor).
    * CSDestructor - No additional members.
    * CSMethod - Has a CSMethodModifier for modifiers[12], a type reference expression for return types and a list of CSTypeParameters (if the method is a template method).
  - CSType - The base class for all types that can be defined by the user. Has an integer for source length. Also supports structs, but they are not supported by Java and therefore not produced by *Sharpen*.
    * CSEnum - Has a list of strings representing all the possible values of the enum.
    * CSTypeDeclaration - Base class for class and interface declarations. Has a list of type reference expressions for base types, a list of members, a list of type parameters (if it is a class/interface template) and a boolean for partial declarations, but this is not necessary in C++.
      · CSClass (CSEnumExtensionClass) - Contains a CSClassModifier for class modifiers[13]. The CSEnumExtensionClass class also has a reference to them enum that it was created for.
      · CSInterface - No additional members. Treated like a class with all methods declared as public and abstract and no fields, as C++ does not have special support for interfaces.
  - CSTypedMember - The base type of members which have a type. Has a type reference expression to represent the type.
    * CSField - Represents a member variable of a class. Adds a list of field modifiers[14] and an initialisation expression, which is optional. If there is a static modifier, then the field needs a definition in the source as well as a declaration in the header.

[10] private, internal, protected, protectedinternal and public
[11] none and static
[12] abstract, abstractoverride, sealed, override, virtual, static and none. Added for C++ only are extern and externstatic
[13] static, abstract, sealed and none
[14] static, readonly, const and volatile

- CSStatement - base class for all statements (a "line of code") and groups of statements. No members.
  - CSBlock - Has a list of statements. Represents a code block, a list of statements (one on each line) surrounded by curly brackets.
  - CSBlockStatement - Represents various keywords which have an expression and a code body.
    - CSForEachStatement - Represents "for *(var : collection)*" syntax. Has a variable declaration object for var. Not supported by C++.
    - CSForStatement - Has a list of initialising expressions and updating expressions. The former are in the first part of the for statement's parentheses, the latter in the last. The inherited expression forms the iteration condition.
    - CSLockStatement - Not supported by C++.
    - CSWhileStatement and CSDoStatement - No new members. Represent the while and do ... while statements.
  - CSBreakStatement, CSContinueStatement - These simply represent the keywords of the same name.
  - CSDeclarationStatement - Contains a single variable declaration. Represents a single, stand-alone variable declaration.
  - CSExpressionStatement - Has a single expression as a field. Represents an expression or tree of expressions. Many standard lines of code will be represented by this type.
  - CSIfStatement - Has an expression to evaluate for truth, and two blocks of code, one underneath the "if", the other underneath the "else" keyword. The latter block is optional.
  - CSReturnStatement, CSThrowStatement - A single expression to return or throw from a method.
  - CSSwitchStatement - Has a single expression to evaluate and list of CSCaseClauses.
  - CSTryStatement - Has a list of CSCatchClauses, a (main) body and a "finally" body. Note that the finally keyword is not supported by C++.
- CSTypeParameter - Used for templates or generics. Has a name and two lists of type parameters: restrictions and sub-parameters.
- CSUsing - Used for using namespace statements. Contains a string with the namespace name.
- CSVariableDeclaration - Represents a variable declaration, either in a statement or in a method declaration. Has a string for the name, an expression for an initialiser (optional) and a type reference expression for the variable type.

# Chapter 4

# Java API

## 4.1 Networking

### 4.1.1 Address resolution

In order to allow Java applications to resolve hostnames into IP addresses, I have reimplemented the java.net.InetAddress, java.net.Inet4Address and java.net.Inet6Address classes. java.net.InetAddress[1] is implemented as an abstract class, from which the other two derive, although almost all of the functionality is implemented in it. It implements the getByName(String), getByAddress(byte[]) and getByAddress(String, byte) methods, the equivalent constructors as well as getters and setters for the address and hostname. The important functionality is in the private native method getIPAddress(String). This method calls a C++ method (via JNI, 2.2.1 on page 4) which performs the DNS lookup. The native method calls the standard getaddrinfo[2] function, which returns an addrinfo[3] structure. If no address is found, the method throws an UnknownHostException, otherwise it examines the ai_family component to determine whether the address is an IPv4 address or an IPv6 address. The ai_addr component is typecasted to either sockaddr_in[4] or sockaddr_in6 as needed then the sin_addr/sin6_addr element is copied to a jbyte pointer. This is then used to copy the address into a byte array, after which the method cleans up and returns. The calling method recognises the address type by looking at the array length and instantiates either an Inet4Address or an Inet6Address object as needed. The Inet4Address and Inet6Address classes override only the abstract addresslength method to return the number of bytes in the address. The Inet4Address class also allows construction with an integer instead of a byte array, for convenience.

[1] For the original class, see `http://download.oracle.com/javase/1.4.2/docs/api/java/net/InetAddress.html`

[2] See `http://msdn.microsoft.com/en-us/library/ms738520(v=vs.85).aspx`

[3] See `http://msdn.microsoft.com/en-us/library/ms737530(v=vs.85).aspx`

[4] See `http://msdn.microsoft.com/en-us/library/ms740496(v=vs.85).aspx`

The implementation here is essentially platform-agnostic, although it has only been tested with Winsock. The only deviation from the standard are the WSAStartup and WSACleanup functions, which must be called before and after using Winsock functions, respectively. The implementation supports IPv6 and should allow it to be used transparently.

### 4.1.2 Datagram sockets

I have also reimplemented the DatagramSocket[5] and DatagramPacket[6] classes. The DatagramPacket class has an address field, port, offset and length fields and most importantly, a data buffer, which uses the ByteBuffer[7] class. The class provides standard constructors as well as getter and setter functions for these fields. The ByteBuffer is allocated with the allocateDirect static method, allowing direct writing to and reading from it in native methods.

The DatagramSocket class, apart from the standard constructors and getter and setter methods, provides wrappers for the socket[8] and bind[9] methods, which are called when the object is constructed. These are required to allow the program to bind to a port number in order to receive packets. Most importantly, it has the send(DatagramPacket) and receive(DatagramPacket) methods. These call the private native methods sendto and recvfrom, respectively. The sendto method obtains a pointer to the data buffer (which is efficient, as it has been allocated as a direct buffer), then initialises a sockaddr structure for either an IPv4 or an IPv6 address, finally calling the sendto[10] function with the appropriate parameters, returning the error number, which is zero if the function is successful.

The recvfrom method also makes use of the direct buffer, calling the recvfrom[11] function, which writes the received data into the buffer, as well as writing a sockaddr structure with the origin of the packet. The recvfrom method allocates a 17 byte array, using one byte to mark whether the packet is an IPv4 or an IPv6 packet. The remaining space is used to store the sender address. After cleaning up, the method returns the number of bytes used. The receive method then uses this to create a new InetAddress object holding the sender address and copies the data to the DatagramPacket specified in the parameter.

[5] See `http://download.oracle.com/javase/1.4.2/docs/api/java/net/DatagramSocket.html`

[6] See `http://download.oracle.com/javase/1.4.2/docs/api/java/net/DatagramPacket.html`

[7] See `http://download.oracle.com/javase/1,5.0/docs/api/java/nio/ByteBuffer.html`

[8] See `http://msdn.microsoft.com/en-us/library/ms740506(v=vs.85).aspx`

[9] See `http://msdn.microsoft.com/en-us/library/ms737550(v=vs.85).aspx`

[10] See `http://msdn.microsoft.com/en-us/library/ms740148(v=vs.85).aspx`

[11] See `http://msdn.microsoft.com/en-us/library/ms740120(v=vs.85).aspx`

## 4.2 Threads

### 4.2.1 Creating new threads

The THREAD[12] class and the RUNNABLE[13] interface have been reimplemented to allow the use of threading in converted code. The RUNNABLE interface is identical to the standard one. The THREAD class supplies a constructor which takes a RUNNABLE object, as well as the START method. This calls the private native START(RUNNABLE) method. This method creates a new PThread (see 2.2.1 on page 4), then stores the RUNNABLE object in a global reference, finally executing the new thread by calling the internal EXECTHREAD function, passing the global reference in a structure. The EXECTHREAD function, now running in the new thread, attaches the thread to the JVM, then calls the RUNNABLE object's RUN method. After this returns, it deletes the global reference, detaches the thread from the JVM and exits. The JNI_ONLOAD and JNI_ONUNLOAD functions are also overloaded, allowing the JVM pointer to be stored to a global variable, as well as initialising the METHODID object needed to make the JNI method call to RUN at load time rather than than execution time, saving overhead.

### 4.2.2 Mutexes and condition variables

In order to allow the use of synchronisation primitives, I have reimplemented the OBJECT[14] class as CPPOBJECT, from which all the other reimplemented classes derive. The class contains one PThreads mutex and one PThreads condition variable, which are initialised as needed by the INITMUTEX and INITCONDITION methods, the pointer to the variables being stored in a LONG value. Likewise, the objects are destroyed in the class finalizer. The class provides wrappers for the PTHREAD_MUTEX_LOCK, PTHREAD_MUTEX_UNLOCK, PTHREAD_COND_SIGNAL, PTHREAD_COND_BROADCAST and PTHREAD_COND_WAIT[15] functions. The methods _ENTER and _EXIT allow for the locking and unlocking of mutexes, while the _NOTIFY, _NOTIFYALL and _WAIT methods supply the same functionality as similarly named OBJECT methods.

[12] See `http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html`

[13] See `http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Runnable.html`

[14] See `http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html`

[15] It should also provide the PTHREAD_TIMEDWAIT function. Unfortunately, MinGW does not support the CLOCK_GETTIME function needed for this to work.

# Chapter 5

# Future Work

In this chapter, I will briefly discuss various aspects of Java-to-C++ conversion that still need attending to. This is by no means a complete list of features missing from the conversion, as there are many ways in which Java and C++ differ which make an isomorphic translation prohibitively difficult.

## 5.1 Synchronisation

In 4.2.2 on the preceding page the CppObject class was described that allows for the use of synchronisation primitives in converted code. The Java language supports the synchronized keyword to allow blocks of code or entire methods to be serialised, preventing concurrency conflicts. The synchronized keyword can make use of any object, as the synchronisation primitives are implemented in the Object[1] class, from which all other Java classes derive. This class has a monitor that is used by the JVM to control access to synchronised code. By using the _enter method of the CppObject at the beginning and the _exit method at the end of such a code block, it is possible to simulate the effect of a synchronized block in C++. The simulation of synchronised methods is more difficult, but may be implemented by having a lockable object for each method, then locking the method by use of a local variable that goes out of scope when the method ends. The great difficulty is how to deal with exceptions properly and prevent deadlock scenarios from occurring. The CppObject class also has the _notify, _notifyAll and _wait methods, which can be used in the same manner as the Java Object methods on which they are modeled. As noted in 4.2.2 on the previous page, timed wait does not work properly yet, although this is only a limitation of the MinGW compiler.

[1] See `http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html`

## 5.2 Collections

Apart from built-in Java arrays which, as discussed in 3.2.3 on page 8, are converted to C++ STD::VECTORs, there is also the matter of Collections to deal with. In general, the use of standard containers is preferred to arrays in both languages. C++ provides a variety of containers through the STL: STD::VECTOR, STD::LIST[2], STD::MAP[3] and STD::SET[4] being the most important. In Java, these are interfaces, namely LIST[5], SET[6] and MAP[7]. These are then implemented by various built-in classes, such as ARRAYLIST[8] or HASHMAP[9]. There are three different approaches to converting the use of Java collections to C++. Firstly, attempt to map the use of Java collections to C++ STL classes, as was done with strings. Unfortunately, the semantics often differ considerably and mapping in general is far less useful than for conversion to C#, where the API is much more similar. Secondly, as was done with networking and threads, it could be attempted to reimplement the classes in native code, giving an efficient implementation which is also a much closer match to the original Java Collections. This is feasible if only a few Collection classes are used, but as a general approach it is time consuming and error prone. Unlike threading and networking, the use of containers does not require any low-level access to the operating system API, only the use of normal language primitives. It is therefore conceivable that an already existing implementation of these containers written *in Java*[10] could be automatically translated to C++. This would ensure that all Collections were available and match the originals semantically. It remains to be seen how efficient this would be, as arrays are much more expensive to create in converted C++ code than normal C/C++ arrays.

## 5.3 Exceptions

Java and C++ both support the use of exceptions, although important differences exist. C++ does not support a FINALLY keyword; this is intentional and there is no likelihood that such a keyword would be added later on[11]. This raises the question how the Java equivalent should be converted to C++. C++ is far more flexible about what can be thrown, Java requires all exceptions to derive from a base class, although this is not an impediment to the correct translation by *Sharpen*. A major problem is compatibility with threading, which was mentioned in 5.1 on the previous page. There is no way to guarantee that a

[2] See `http://www.cplusplus.com/reference/stl/list/`
[3] See `http://www.cplusplus.com/reference/stl/map/`
[4] See `http://www.cplusplus.com/reference/stl/set/`
[5] See `http://download.oracle.com/javase/1,5.0/docs/api/java/util/List.html`
[6] See `http://download.oracle.com/javase/1,5.0/docs/api/java/util/Set.html`
[7] See `http://download.oracle.com/javase/1,5.0/docs/api/java/util/Map.html`
[8] See `http://download.oracle.com/javase/1,5.0/docs/api/java/util/ArrayList.html`
[9] See `http://download.oracle.com/javase/1,5.0/docs/api/java/util/HashMap.html`
[10] For instance Apache Harmony, see `http://harmony.apache.org/`
[11] Bjarne Stroustrup states this explicitly in his FAQ: `http://www2.research.att.com/~bs/bs_faq2.html`

mutex is unlocked if an exception is thrown between the locking and unlocking of a mutex, meaning that a deadlock could occur. As exceptions are widely used in Java code, this must be addressed before such code can safely be used in a multi-threaded environment. It may be possible to simulate the FINALLY block using labels and GOTO statements, although this could result in ugly and incomprehensible code.

## 5.4 C++ 2011

Finally, I would like to refer to the *C++ 2011* standard. Many of the issues that arose during this semester thesis could have either been prevented entirely or greatly simplified by the new standard[12]. In particular:

- The use of initialiser lists, so that temporary variables are no longer needed for array initialisation. The usage is now the same as with Java arrays and containers.
- The STD::ARRAY container, which in particular stores its size and it is a much closer match semantically to Java arrays than STD::VECTORs which can grow or shrink. It may also be more light-weight.
- Built-in support for the foreach statement, with the same syntax as Java and C#.
- In-declaration member initialisation, so it is no longer necessary to create a new default constructor to define member variables.
- Typed null pointers with the NULLPTR keyword.
- Methods can be explicitly declared to be overridden, and it is possible to prevent methods from being overridden (with the new FINAL keyword).
- UNIQUE_PTR and SHARED_PTR, which are especially useful for ensuring that objects are deleted in case of exceptions. They may reduce the need for garbage collection.
- Possibly the most important improvement is the addition of threading to the standard library, which has many similar semantics to Java threads. This would allow direct translation of Java threads to C++ threads without worrying about compatibility or exception issues.

[12] See `http://www2.research.att.com/~bs/C++0xFAQ.html`

# Chapter 6

# Conclusion

In this semester thesis, I have developed a platform to convert Java to C++ code, with a view to simplify porting the *Pulsar* distributed media streaming system to new platforms. To accomplish this I have adapted the Eclipse plugin *Sharpen*, which converts Java code to C# code, to allow it to generate C++ code as an output, and I have reimplemented parts of the Java API in C++ using only generic APIs to maximise portability.

The *Sharpen* plugin essentially consists of two visitors, the first of which steps through the Java source AST and creates a new AST for C#, the second uses this second AST to output C# code. I have modified the first visitor and replaced the second with two new visitors, producing a C++ class header and a source file. I have made changes to the code output, taking into account the syntactical differences between the languages. In particular, I have adapted the handling of arrays, strings and enumerations, classes with inheritance and interfaces as well as namespaces and compilation units. The plugin produces compliant code for a large subset of the Java language.

Furthermore, I have translated parts of the Java API to C++, for usage by the converted code. I have paid special attention to threading, with the creation of new threads and synchronisation, as well networking, with hostname-to-IP resolution as well as UDP based transport enabled. This code also supports IPv6 and is agnostic of the layer 3 protocol used.

However, the support of C++ is by no means complete, nor was this the objective of this thesis. In particular, aspects not touched upon include exceptions, collections as well as the implementation of the SYNCHRONIZED keyword. Even then, not all valid Java code can be translated, as C++ lacks many elements present in Java and to some extent also in C#. It is clear that further work will be needed, especially with respect to the API, in order to translate large and complicated programs such as *Pulsar*.

In a final section, I have pointed out the advantages of the *C++ 2011* standard, which adds many new features to the language and to the API that makes conversion both considerably less arduous, more semantically accurate and also shorter and more elegant.