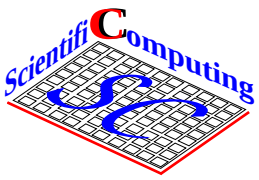
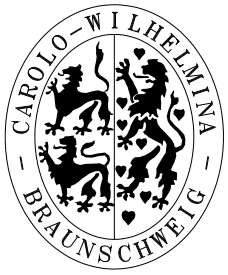


The Component Template Library Protocol and its Java Implementation

Project Work

Boris Bügling

Institute of Scientific Computing
Technical University Braunschweig
Brunswick, Germany



Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Braunschweig, den 30th March 2006

Boris Bügling

Contents

1 Introduction **1**

2 Related Work **1**
2.1 Remote Method Invocation . . . 1
2.2 CORBA . . . 2
2.3 Java RMI . . . 2
2.4 Microsoft .NET . . . 2
2.5 SOAP . . . 3
2.6 Ice . . . 3
2.7 CCA . . . 3
2.8 XML-RPC . . . 3

3 Resources **4**

4 Results **4**
4.1 The CTL Protocol . . . 4
4.1.1 Overview . . . 4
4.1.2 Limitations . . . 4
4.1.3 Terminology . . . 5
4.1.4 Fundamentals . . . 5
4.1.5 Composites . . . 5
4.1.6 CTL-specific types . . . 6
4.1.7 User-defined types . . . 7
4.1.8 Communication . . . 8
4.1.9 Header . . . 9
4.1.10 DAT messages . . . 10
4.1.11 Remote call (RMI) . . . 10
4.1.12 Remote answer . . . 11
4.1.13 Exceptions . . . 12
4.1.14 Remote interfaces . . . 12
4.1.15 Resource manager . . . 16
4.2 Using the CTL4j . . . 16
4.2.1 Installation . . . 16
4.2.2 Implementing the calculator . . . 16
4.2.3 Implementing the client . . . 17
4.2.4 Implementing the server . . . 18
4.2.5 Using the distributed system . . . 18
4.3 Example: A distributed Dijkstra algorithm . . . 20
4.3.1 Dijkstra's algorithm . . . 20
4.3.2 Serializing a graph . . . 20
4.3.3 The algorithm itself . . . 31
4.3.4 Implementing the client . . . 33
4.3.5 Implementing the server . . . 34
4.3.6 Using the distributed system . . . 34

5 Method **34**
5.1 Initial design . . . 35
5.2 Implementation . . . 35
5.2.1 ReflWrap . . . 35
5.2.2 CodeGen . . . 36
5.2.3 CTL . . . 36
5.2.4 Python . . . 39

6 Discussion **39**
6.1 Function Call Overhead/Performance . . . 39
6.1.1 Discussion . . . 40
6.2 Protocol . . . 41
6.3 Comparison of distributed object frameworks . . . 41
6.3.1 CTL/C++ . . . 41
6.3.2 CORBA . . . 42
6.3.3 Microsoft .NET . . . 44
6.3.4 Java RMI . . . 45
6.3.5 SOAP . . . 45

7 Conclusion **46**

A Component interface grammar **47**

B Obtaining the CTL **48**

List of Figures

1 CTL layers 8
2 Basic structure 9
3 Remote call structure 10
4 Remote answer structure 11
5 Exception hierarchy 12
6 perf performance. 39
7 CORBA ORB architecure 42

List of Tables

1 Header details 9
2 Benchmark results 40

Listings

1 IDL for a simple calculator . . . 2
2 Java RMI interface declaration . . . 2
3 .NET remoting interface declaration . . . 3
4 SOAP interface declaration . . . 3
5 Array . . . 5
6 String . . . 6
7 Tupel . . . 6
8 Writable interface . . . 7
9 User-defined data structure . . . 7
10 Reserved tags . . . 9
11 DAT methods . . . 10
12 RMI call . . . 11
13 RMI answer . . . 11
14 Receive exception at group level. . . . 12
15 CI example 1 . . . 13
16 Implementation of add() . . . 13
17 CI example 2 . . . 13
18 CI example 4 . . . 14
19 Excerpt from the generated Java CI . . . 14
20 CI example 4; generated from CI example 1 . . . 14
21 The Example, implemented in Python . . . 15
22 py2c XML example . . . 15
23 CI for the resource manager . . . 16
24 Calculator interface . . . 16
25 Calculator client . . . 17
26 Calculator service . . . 18
27 Ant build.xml . . . 18
28 CTL.Types.Node . . . 20
29 CTL.Types.Tupel . . . 23
30 The ReflWrap.TemplHack interface . . . 27
31 CTL.Types.Graph . . . 27
32 Dijkstra interface . . . 31
33 DijkstraRI client . . . 33
34 SerialIn methods . . . 36
35 Example of a serialRead() call . . . 37
36 The ReflWrap.TemplHack interface . . . 37
37 SerialOut methods . . . 37
38 Special messages . . . 38
39 ObjectMap methods . . . 38
40 call a function with this signature . . . 39
41 Example: defining an array of int4 . . . 42
42 Component interface grammar . . . 47

Abstract

The Component Template Library (CTL) can be used to realize distributed component based software systems. This document describes and discusses the CTL protocol, shows how to use it for rapid development of distributed software projects and compares it to similar systems like CORBA and the CCA.

1 Introduction

Many modern applications are in need of a decent distributed object framework, such as CORBA or DCOM (Distributed Component Object Model, see [Mic]). Most of the existing solutions share the problem that they dump a significant amount of work on the application programmer and that they enforce a strict separation of distributed and traditional monolithic systems. These problems are addressed by the CTL C++ implementation (CTL/C++), which tries to make the development of distributed systems as easy as possible.

This paper describes the design and structure of the CTL protocol itself and the Java implementation (CTL4j). By describing a distributed Dijkstra algorithm application, it will show how to use the CTL for your own software projects. Finally, the protocol is discussed and compared with other existing solutions, especially in its usability and performance. The document also serves as an introduction into remote method invocation (RMI) in general and how the major distributed object frameworks work in practice.

If you are only interested in using the CTL for your own application, go right to the section 4.2 of this paper.

Note: For the sake of readability, all CTL4j methods have the list of thrown exceptions stripped in the source code snippets. Use the CTL4j API documentation for information about exceptions.

2 Related Work

2.1 Remote Method Invocation

In general, RMI is a mechanism which allows programs to make remote function calls and access remotely stored objects. The communication happens over a serialized byte stream, which can, for example, be transported with TCP/IP, between a client (the one who calls a method) and a server (the one who will execute it). Both sides have to share their knowledge about available classes, functions and methods, this is done in a remote interface (RI), sometimes in a special language, like the interface definition language (IDL) of CORBA. Such a collection of related classes and functions and the way to interact with them is usually called a component. Ideally, the client-side application should not need to know if a certain component is available locally or will be invoked remotely. Of course, this means that there needs to be an authority which can provide information about available components to distributed applications, an example for this is the Oject Request Broker (ORB), which handles Interoperable Object Reference (IOR; basically an URL for an object) of available components (see 4.1.15 for the CTL protocol's solution to this problem).

The exchange of structured data types over a serialized stream has to be abstracted from their binary representation. In the CTL protocol, any complex data structure is a composition of simpler types, which can either be a fundamental type or one of

a limited number of composites and to read a data structure from a stream only the binary representation is needed. The available fundamentals and compositions of the CTL protocol will be discussed later (see 4.1.4).

In the following, several available distributed object solutions and their usage will be described, see 6.3 for a comparison of them with the CTL.

2.2 CORBA

The Common Object Request Broker Architecture (CORBA) is a distributed object specification by the Object Management Group (OMG). It has its own communication protocol called IIOP and a special languages for defining interfaces, called Interface Definition Language (IDL). There are implementations available in many programming languages. For this paper, I used ORBit2/libidl2, the open-source CORBA implementation of the GNOME project.

Listing 1: IDL for a simple calculator

```
interface Perf
{
        typedef sequence<double> doublearr;
        doublearr send (in doublearr a);
};
```

2.3 Java RMI

Java RMI is a part of the standard Java SDK by Sun. It uses Java interfaces as definition language of the remote interfaces and is therefore limited to be used by Java applications only. It has been available since JDK 1.02 and is similar to CORBA in its complexity.

Listing 2: Java RMI interface declaration

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Perf extends Remote
{
        double[] send (double[] a) throws RemoteException;
}
```

2.4 Microsoft .NET

Microsoft's .NET framework offers two ways for distributed components. The first one uses SOAP as communication protocol and is meant for writing web services with .NET and should be compatible with SOAP implementations in other programming languages. The other one is called Remoting and provides a complete infrastructur for distributed objects. It uses a binary stream for communication. Interfaces for Remoting applications are written in C# and can therefore only be used by programming languages ported to the .NET runtime. The relatively new component architecture of .NET will replace the formerly used DCOM on the Windows platform. As in the CTL4j, there is no explicit interface declaration, it is declared implicitly by the implementation.

Listing 3: .NET remoting interface declaration

```
using System;

namespace Bench
{
        public interface Perf
        {
                double[] send (double[] a);
        }
}
```

2.5 SOAP

The Simple Object Access Protocol (SOAP) uses the HTTP protocol to send and receive XML messages. It is a recommendation ([Grob]) by the W3C and there are numerous implementations for most modern programming languages. The SOAP example code for this paper was developed using C and gsoap2 (see [vE]).

Listing 4: SOAP interface declaration

```
typedef struct
{
        double[100] foo;
} doublearr;

h__send(doublearr, doublearr*);
```

2.6 Ice

The Internet Communication Engine (Ice) (see [MW05]) is marketed as a modern alternative to CORBA by its author, the company ZeroC Inc (see [Incb]. Like .NET Remoting, it is a proprietary protocol, but its source code is available under the GPL. It has support for several languages, namely C++, Java, C#, Visual Basic, Python and PHP.

2.7 CCA

The Common Component Architecture (CCA) is a fairly new high-performance and distributed computing framework. It was developed in December 1998 by US national energy laboratories and research laboratories from the Universities of Utah and Indiana and was sponsored by the Department of Energy (DOE). Unlike the other approaches, the CCA is focused on scientic computing and provides pre-built components for it. Furthermore, it defines a standard for communication and component retrieval, as well as the Scientific Interface Definition (SIDL) for defining component interfaces, but it does not enforce a specific implementation. However, the CCA forum ([For]) provides a reference implementation with SIDL language bindings for C, C++, Fortran, Java and Python.

2.8 XML-RPC

XML-RPC is a standard for simple remote procedure calls over TCP/IP networks using HTTP as transport and XML as encoding (see [Inca]). As it does not provide support

for using distributed components, no naming services and neither location nor access transparency, it will not be included in the 'Comparison' section (see 6.3).

3 Resources

The Java implementation of the CTL (CTL4j), which is the base of this paper, was derived from the original C++ implementation (see [Nie05] by Dr. Rainer Niekamp. It was developed on Linux using the Sun Java 5.0 SDK and the Java Secure Channel library (see [JCr]), which is needed for the SSHv2 communication. The build system uses Apache Ant, which is also a requirement for building the CTL4j itself or applications which use it. The interface definition language compiler CTLcc consists of two parts, a Java implementation, which generates C++ interfaces from Java class files using Reflection and a Python script which generates Java interfaces from C++ component interface definitions. The script ctlcc.py uses the module *pyparsing* (see [McG]) for a lex/yacc-like parser of the CTL grammar. Due to a bug in Java's *Reflection* API for inspecting generics, the Jakarta ByteCode Engineering Library (BCEL) was used for parsing the bytecode in some situations. This is handled transparently by *ClassInfo* objects and cannot be seen when reading the CTL4j or code generator code itself.

4 Results

4.1 The CTL Protocol

4.1.1 Overview

The CTL protocol has two main goals:

- Providing a lightweight and simple protocol which can be used on top of several communication methods (TCP/IP, MPI and others).
- Making the process of writing an application or a service which uses the CTL protocol as transparent as possible. The developer of a service can write his implementation like he would write a normal local class, with the exception that he needs to give the CTL a method to serialize the contained data (in the Java implementation this is as simple as implementing the *Writable* interface). The developer of a client only needs to know how he can choose a service within the CTL API and how he starts (and for the CTl4j, stops it; both done by one single command), he can use the objects provided by CTL services as if they were standard local objects.

4.1.2 Limitations

Compared to the C++ implementation, CTL4j has a few shortcomings: There is no support for communication in groups with more than two participants. As of now, there is no support for exceptions, other than the CTLException which occurs when a requested class is not available at the other end. In addition to that, there are some limitations in terms of using non-constant arguments, which will be described later.

4.1.3 Terminology

- **Client**
  The application which initializes the communication, either through starting the service via ssh or by making the initial connection.
- **'Magic' string**
  To ensure that the communication stream is not corrupted, the hexadecimal number `1F3E:A28E:2CF0:9378:AA01:0744:5D31:710A` is sent during the handshake.
- **Service**
  The provider of a component implementation which is called by the client.

4.1.4 Fundamentals

Every complex data structure can be broken down to a few fundamental data types and compositions of them. This section will talk about the available types in CTL4j, their CTL/C++ equivalent and the handling of user-defined types. As there are no unsigned types in Java, the unsigned CTL/C++ types will be mentioned with their equally sized signed counterparts. In the rest of this paper, the CTL4j types will be used.

Integer types

boolean (bool): An one byte integer with values in $\{0,1\}$.
char (char, ctl::uchar): An one byte integer, usually interpreted as an ASCII character.
N/A (ctl::int1, ctl::uint1): An one byte integer.
short (ctl::int2, ctl::uint2): A two byte integer.
int (ctl::int4, ctl::uint4): A four byte integer.
long (ctl::int8, ctl::uint8): An eight byte integer.

Floating-point types

float (ctl::real4): A four byte floating-point number.
double (ctl::real8): An eight byte floating-point number.

Other types

void (void): An empty type.

4.1.5 Composites

Arrays

An array is an arbitrary-length list of variables of one type. In Java, the standard notation with a postfix of a left and a right bracket (*[]*) is used, whereas in C++, it is a template *array<T>*. It is serialized as a *long*, the length of the array, and all the elements.

Listing 5: Array

```
array<T>                size, T, ...
        for (i < size)
                istream << array[i]
```

String

A string is an arbitrary-length list of variables of one type, terminated by a single null byte. In both Java and C++ the standard string types are used, namely *java.lang.String* and *std::string*. A string is serialized as the elements followed by a single null element.

Listing 6: String

```
cstring<T>              T, ..., 0
        while (T)
                istream << cstring[i]
```

Tuple
A tuple is a fixed-length list of variables of different types. In both languages, it is a template class, called *CTL.Tuple* in Java. A tuple is serialized by simply serializing all types. Basically, a tuple is equivalent to a *struct*.

Listing 7: Tupel

```
tuple<T0, ..., Tn>
        T0;
        ...
        Tn;
```

Reference
A reference is used for serializing data structures which can contain multiple pointers to one element, for example a linked list or a graph. For saving bandwidth and avoiding infinite loops (imagine a cyclic graph being send which each element copied to the stream), each element is sent only once and an ID is sent for all subsequent occassions. This type exists as a template class in both implementations, being called *CTL.Types.Reference* in CTL4j. It is serialized as a boolean, which is set to *true* for the first occassion and to *false* for all others, followed by an long, an unique type ID (*long*), the binary size of the element (*int*) and its serialized content, respectively its unique type ID. An example for using this type will be given in section 4.3. The *Reference* type is also used for reading polymorphic types in the CTL/C++, as its reader only knows base types.

4.1.6 CTL-specific types

Any The *any* (ctl::any in C++) type can be used to serialize a type in a manner that the receiver does not have to know which type will be read. It is implemented as an Annotation in the CTL4j, which marks that a certain parameter will not be send directly, but as an *any* object. It is serialized as a string (an empty string for a null object), followed by an int, which represents the size of the payload and the serialized data itself. For compatibility with ctl::any, the type *CTL.CCompat.AnyObj* exists.
FID A *FID* (FunctionID) is an unique identifier for a function or method. It consists of a short, the numerical ID, and a string, the function name. which is empty for a method. For static methods and constructors, it is a fully-qualified name, consisting of the method's namespace (or package in the Java terminology), followed by ::, the name of the class. a single : and the numerical ID, either with the postfix *C* for a constructor or *S* for a static method.
GroupInfo A *GroupInfo* is used to hold information about group members and send them during the handshake. It contains the *PeerID* of the process and a *rPointer* to its *Group* object.
IPaddr There is only one type for both IPv4 and IPv6 network addresses in the CTL, the `IPaddr`. It is an array of 8 shorts, which are all used for the 128bit IPv6 addresses, respectively the first 6 shorts set to zero and the last two representing the 32bit IPv4 addresses.

Location A *Location* (ctl::location) holds all necessary parameters for reaching a component. These are the name of an executable[1], its path in the filesystem, a hostname, a port and login, usually username and password. A valid *Location* is needed to initialize communication, it can either be set explicitly by the application programmer or obtained by a resource manager (see 4.1.15). It is serialized as one string which contains all the information. Optionally, the password can be left out and you will be prompted at runtime for it. The C++ location strings are only partially supported by CTL4j as of now.
ObjectID A single long is used as identifier for objects. In Java, this is the result of the method *hashCode()*, whereas it is a pointer in C++. See section 5.2.3 for more information about the internal storage and retrieval of these IDs.
PeerID The Address of a peer, which consists of an int and an *IPaddr*. It can either be a physical address, port and IP address, or a logical, GroupID and the zero IPaddr.
rPointer A *rPointer* is a remote pointer to an object, which contains the *PeerID* of the host the object lives on, its *ObjectID* and an int, the reference counter.
rResult A *rResult* is used to receive and store results of a call (see 3). It is either handled internally by the CTL, enabling application programmers to transparently use remote methods exactly like local methods, or it can be handled by the application itself using the alternative methods with the *__rr* suffix. The latter can be used to execute numerous function calls in parallel and obtaining their results from the rResults later on.

4.1.7 User-defined types

Being a flexible application framework, the CTL of course also supports arbitrary user-defined types, which are, as explained above (4.1.4, nothing more than a composite of simpler types, all the way down to the fundamentals (4.1.4). To be able to use a new data structure, one has to implement the *Writable* interface:

Listing 8: Writable interface

```
public interface Writable
{
        void read (SerialIn in);
        void write (SerialOut out);
}
```

Those two methods are used by the CTL4j streams (see 5.2.3) to read and write the types, which means that they have to match each other to work properly. The streams or composite classes (see 4.1.5 offer read/write functions for all types mentioned above, which are listed in the accompanying API documentation for CTL4j. A simple user-defined data-structure, consisting of a *double* and a *float* would be implemented like this:

Listing 9: User-defined data structure

```
import CTL.*;

public class DoubleDash implements Writable
{
        private double d;
        private float f;
```

[1] CTL4j uses a script which takes care of setting up the classpath and starting the virtual machine

```
        public void read (SerialIn in)
        {
                d = in.readDouble();
                f = in.readFloat();
        }

        public void write (SerialOut out)
        {
                out.writeDouble(d);
                out.writeFloat(f);
        }
}
```

Some more sophisticated examples will be shown later on (section 4.2 and 4.3.

4.1.8 Communication

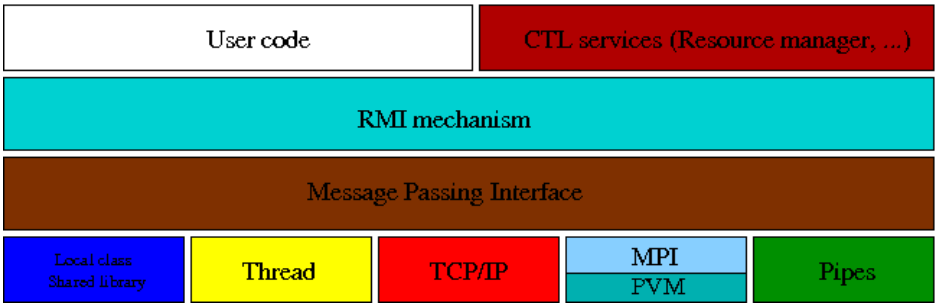


Figure 1: *Shows the layer concept of the CTL protocol*

As shown in the diagram, the CTL protocol consists of three communication layers, the underlying communication protocol, such as TCP/IP, the message passing interface and the RMI mechanism which lies on top. Above that lies the application layer, which consists of additional CTL services and all user applications and components.
The first step for establishing a communication is a handshake. The contacted service first sends an empty package of the type DAT, which has the IP set to the 'magic'-string and tells the client on which port the service listens. Now the client will connect to this port and send a DAT package with its own GroupInfo object as payload, the server will reply with its GroupInfo object and the communication is established.
The protocol consists of two basic operations, a `remote call` (see Figure 3), which sends the request for a method and a `remote answer` (see Figure 4.1.12), which sends the results of a call, return values, possible modified arguments and/or an exception, back to the caller.
The connection between client and service is persistent and will only be closed when the client requests its termination or a fatal error occurs. If the service was started directly by the client, it will be terminated together with the client.

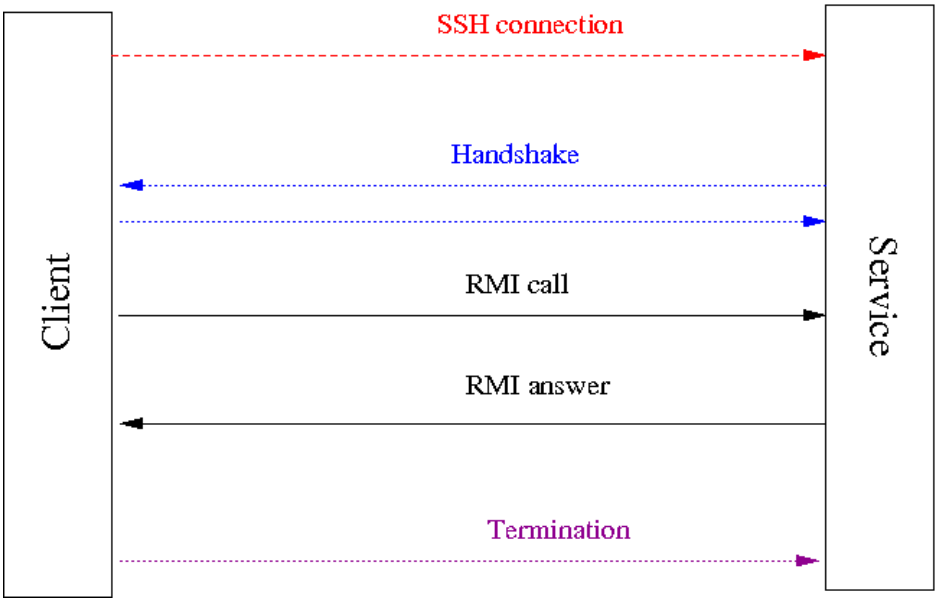


Figure 2: *Shows the structure of communication via the CTL protocol*

4.1.9 Header

All messages passed via the CTL protocol consist of a fixed size header and a variable sized payload. The header is structured like this:

Table 1: Header details

Payload size (*long*)	8 byte
Message tag (*int*)	4 byte
PeerID (see 4.1.6)	20 byte
LogicalID (*int*)	4 byte

The CTL4j has wrapped assembling, reading and writing in the class *CTL.Types.Header*. The message tag is used differently by the various message types, however, there are some reserved tags (defined in *CTL.Remote*):

Listing 10: Reserved tags

```
public final static int EOC = -1;
public final static int DAT = 1;
public final static int OPER = 2;
public final static int RMI = 3;
public final static int CTRL = 4;
public final static int ERR = 5;
public final static int UNDEF = 255;
```

Tags with a value greater than 255 serve for remote answers (section 4.1.12).

4.1.10 DAT messages

A *DAT* message can be used to send data to other processes. This happens synchronized and both communication partners have to know what type is being sent. *CTL.Remote* defines two methods for this:

Listing 11: DAT methods

```
public static Object readDAT (Communicator comm, String type);
public static void writeDAT (Communicator comm, Object data);
```

4.1.11 Remote call (RMI)

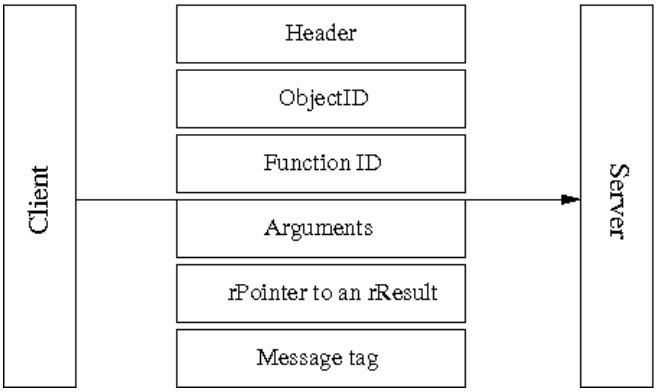


Figure 3: *Shows the structure of a RMI call*

A remote call serves as request for the remote execution of a function or method. This mechanism is used transparently by the CTL implementations, so that application programmers just see a normal method invocation, whereas the language-independant RMI, which is described in the following, is used.
As any message, the remote call starts with the CTL protocol's header (see 4.1.9) with the message tag set to *RMI*. The following *ObjectID* is either zero, for constructors and static methods, or the ID of an already allocated object. Each instance of a remote object (derived from the base class *CTL.RI*) just acts as a *rPointer* with the real object stored on the remote side, where the ID is sent back by a remote answer (see 4.1.12) after calling its constructor for the first time. To specify which method should be invoked, the *FunctionID* (see 4.1.6) is used. The arguments are all serialized as described above and can be deserialized by the communication partner, because both share the same remote interface declaration (see 4.1.14). The next two data fields are needed to be able to receive results after the method invocation. The method wrapper generated by *CodeGen* (see 5.2.2) allocates a new *rResult* object (see 4.1.6) and sends a *rPointer* of it. The receiving end will know to whom it has to send the results (return value and modified arguments[2]), but to be able to retrieve specific result messages from a queue,

[2]Note: The Java programming language always uses function arguments by value, which means that the only way to modify an argument is manipulating an object (for example by *setFoo()* methods). However, modified arguments can still be read by the caller by using the *_rr() functions, which return the *rResult* object directly.

the caller appends a message tag (*int*) with a value of greater than 255, which will later be used as the answer's tag in its header.
Sending remote calls is handled by the class *CTL.Remote* like this:

Listing 12: RMI call

```
public static void call (Communicator comm, Header head, long
    objID, FID fid, IStream2 args, rPointer objID2);
```

This method handles both the remote invocation and configurable debugging/logging with *CTL.Logger* (see 5.2.3).

4.1.12 Remote answer

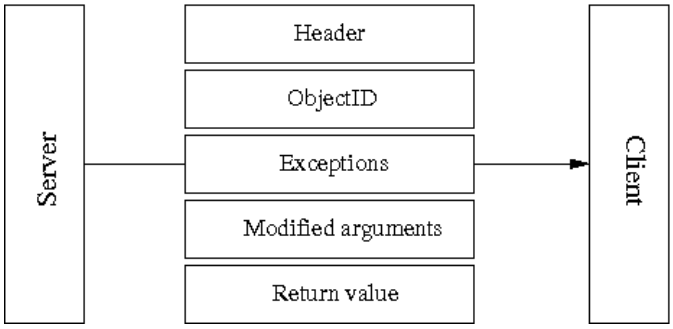


Figure 4: *Shows the structure of a RMI answer*

Most RMI calls will not be procedures, but functions which will return some kind of results, either as an exception or the return value of the function, as well as possibly modified arguments[3]. Those answers will be send through the *answer()* method. If all arguments are constant, the return value is void and no user-defined exceptions exist, the method will send no answer back.
As any CTL message, the answer starts with a Header, with the message tag set to the unique tag sent by the call (see 3), followed by the ObjectID of the Object to which the called method belongs (0 for static methods/functions). The encoding of exceptions is described in the next section (4.1.13). All arguments which are not marked as const, will be sent back, because the caller cannot know which were modified by the call and which were not. Both the return value and the arguments are serialized as described earlier (see 4.1.4). During development with the CTL4j, programmers will usually not have to deal with *answer()*, because it is used inside the generated RI classes and not in their own code.
Sending remote answers is handled by the class *CTL.Remote* like this:

Listing 13: RMI answer

```
public static void answer (Communicator comm, Header head, long
     objID, Except ex, IStream2 args);
```

As *call()*, this method handles the logging with *CTL.Logger* automatically.

[3] See for more information about Java's handling of this.

4.1.13 Exceptions

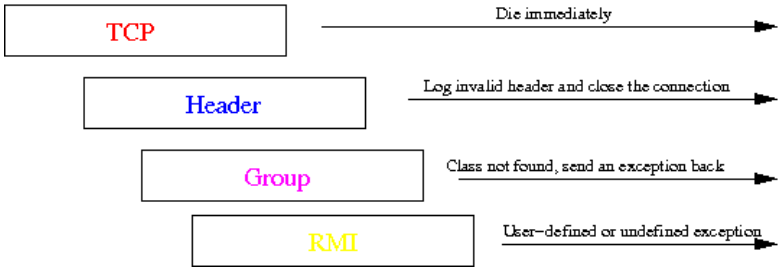


Figure 5: *Shows the hierarchy of possible exceptions*

There are several ways of reacting to exceptions in the CTL protocol, depending on their position in the hierarchy, as seen below. Exceptions on TCP or Header level will not be forwarded in any way, they just lead to the termination of the connection. Examples are: corrupted headers and socket exceptions.
Exceptions at Group level will lead to a call to a special static method defined in *CTL.Group*:

Listing 14: Receive exception at group level.

```
@builtin @sFID(4) public void recvException (String msg);
```

The reason for this exception is usually that a class was not found. The message will be logged by the receiving end and the communication for this call ends. The annotation *builtin* is used to mark methods which will not send any answers back, like *recvTermination()*.
All other possible exceptions occur at RMI level, they are the ones seen in the *answer()* diagram above (see 4.1.12). If no exception was thrown by the call, a single null byte is send back. Other exceptions are usually send as message string, but they are basically user-defined *AnyObjs* (see 4.1.6), so any kind of object may be sent back as exception. As of now, CTL4j has no support for user-defined exceptions.

4.1.14 Remote interfaces

Every CTL component is an implementation of a component interface (RI; also Component Interface, CI). It has the same function as the CORBA IDL, being a simple formal description of which classes, functions and methods a component provides, from which the stubs, or in the CTL case the complete functions which perform data serialization and transport, are generated. The CI grammar can be found in the appendix (A).
Depending on which language you use for developing your CTL application, there are different methods of generating or writing a CI. A simple component with one method *add()* will be used as an example in this section.

- C/C++

  Usually, you will write the CI for your component by hand. For the example, that will look like this:

Listing 15: CI example 1

```
#ifndef __ADD_CI__
#define __ADD_CI__

#include <ctl.h>

#define CTL_Class AddCI
#include CTL_ClassBegin
        #define CTL_Method1 int4, add (const int4, const
            int4), 2
#include CTL_ClassEnd

#endif // __ADD_CI__
```

As you can see, preprocessor macros are used for defining the RI. The class *AddRI* has one method *add()* with two constant integers as argument and one as return value. Due to the fact that the C preprocessor (cpp) cannot count lists, the number of arguments needs to be appended to each declaration. See [Nie05] for a more detailled description of C++ CIs.

If you already have a working implementation of your component and you just want to use it in a distributed application using the CTL/C++, you can use the ri-generator written by Oliver Pajonk, which is written in Java using JavaCC. It parses existing C/C++ headers and generates a CI for them. You might have a C implementation of *add()* which looks like this:

Listing 16: Implementation of add()

```
#ifndef __ADD_H__
#define __ADD_H__

int add (const int a, const int b);

#endif // __ADD_H__
```

Because C does not have semantics for defining classes, a library Add will be generated with that function:

Listing 17: CI example 2

```
#ifndef __ADD_CI
#define __ADD_CI

#include <ctl.h>

#define CTL_Library Add
#include CTL_LibBegin
#define CTL_Function1 int, add, (const int /*a*/, const int
     /*b*/), 2
#include CTL_LibEnd

#endif // __ADD_CI
```

- Java

Due to the lack of a preprocessor in Java, the CTL4j has a code generator which uses *Reflection*. This means that the communication code is directly generated from a Java implementation. Our example looks like this in Java:

Listing 18: CI example 4

```
import CTL.Annotate.*;
import ReflWrap.*;

public class AddRI
{
        public int add (@const_ int arg0, @const_ int arg1)
        {
                return arg0 + arg1;
        }
}
```

Due to Java's lack of a *const* keyword[4], the Annotation *const_* is used for marking constant arguments. For the sake of simplicity, the annotation is also defined for classes, meaning that all arguments of all methods of that class are constant (see 3 for a discussion of non-const arguments in Java). Apart from this, the implementation is just a normal Java class, the rest of the work is done by the code generator for you[5]. A look into the generated file reveals that two methods are generated for each method in the source class:

Listing 19: Excerpt from the generated Java CI

```
public int add (Integer arg0, Integer arg1);

public rResult add_rr (Integer arg0, Integer arg1);
```

The *add()* method can be used like a method of a locally available class, but the *add_rr()* returns a *rResult* (see 4.1.6). The returned object can be used if you want to call several methods at once and catch the results later for example[6].

While this method is fine for developing new Java components or use existing code in distributed applications, nobody wants to write interface stubs for C++ classes manually. For compatibility with C++, two parsers are available: *ctlcc.py* which is written in Python and converts C++ CIs to Java stubs and *CTL.CCompat.CTLcc* which generates CIs for existing Java components using *Reflection*. Below is an example for code generated by the Python parser, from which *CTLcc* can generate the original C++ CI. The *sFID* annotation is used to assign a static Function ID to a method for compatibility[7].

Listing 20: CI example 4; generated from CI example 1

```
import CTL.Annotate.*;
import ReflWrap.*;

public class AddRI
{
```

[4] Although it is a reserved keyword for the compiler, that is why the underscore is used
[5] With more than 500 lines of code, the generated RI is too long to be published here
[6] Example code can be found in the CTL4j distribution.
[7] By default, FIDs are assigned automatically at runtime

```
        @sFID(1) public int add (@const_ int arg0, @const_
            int arg1)
        {
                return -1;
        }
}
```

- Python

  As described earlier (5.2.4), JPype and Swig allow Python applications to use existing Java and C++ components, because there is no implementation of the CTL protocol in that language. By using the C interface (libpython), users may also write components directly in Python with the help of *py2c*[8]. By using the *Parser* class, which is distributed with Python and gives developers access to the internal parser of the interpreter, an existing class can be parsed. A C++ wrapper for that class is then generated which can be used to generate a CI using the above mentioned ri-generator.

  Listing 21: The Example, implemented in Python

```
class Add:
    def __init__ (self):
                pass

        def add (self, a, b):
        """ @param a int
                @param b int
                @return int """
                return a+b
```

  Because of Python's dynamic typing, the user has to specify which types are used by his methods. For this, py2c offers two mechanisms:

  – Putting the definition in a docstring directly in the classes code, like in the example. A doxygen-inspired syntax is used for that declaration.
  – Providing an external XML file with the type information:

    Listing 22: py2c XML example

```
<py2c>
        <method name="add">
                <param name="a">int</param>
                <param name="b">int</param>
                <returns>int</returns>
        </method>
</py2c>
```

  No matter which method is used, a C++ wrapper and a corresponding header file is generated for accessing the class from C++.

[8] As of now, py2c is still experimental.

4.1.15 Resource manager

For true transparency, distributed applications need a service which provides them with available implementations of an interface and their locations. The CTL itself only defines an interface for talking to such managers, but leaves the implementation at user-level.

Listing 23: CI for the resource manager

```
#define CTL_Class CTL_Locator
#include CTL_ClassBegin

        #define CTL_Constructor1 (const cstring<char> /*
            filename*/), 1
        #define CTL_Method1  string, operator(), (const string
            /*CI_Type*/, const any /*property*/) const , 2
        #define CTL_Method2  string, get, (const cstring<char>
            /*CI_Type*/, const any /*property*/) const , 2

#include CTL_ClassEnd
```

Based on a string containing the name of the class or library to look for and a property, which is an any and can therefore also be defined by the coder of the resource manager, a ctl::location is determined and returned to the caller. As of now, the CTL4j cannot use a resource manager.

4.2 Using the CTL4j

This section will walk you through the implementation of an easy distributed calculator, which can add and subtract two integer numbers, in Java. It covers both the installation of CTL4j and all implementation details, but basic Java programming experience is required.

4.2.1 Installation

Obtain a CTL4j tarball and see section 3 for all prerequisites. You can compile it with *make* then and run the testsuite with *make check* or generate the API documentation with *make doc*. For the faint of heart, there is a JAR available, which just has to be in *$CLASSPATH*.

4.2.2 Implementing the calculator

As you know from section 4.1.14, the implementation class will also act as interface definition for the remote communication. The simple calculator component will only be able to add and subtract two integer numbers. Those methods will be static with constant arguments:

Listing 24: Calculator interface

```
import CTL.Annotate;

package Impl;

public class Calc
{
```

```
        public static int add (@const_ int i, @const_ int j)
        {
                return i + j;
        }

        public static int sub (@const_ int i, @const_ int j)
        {
                return i - j;
        }
}
```

As you can see, it looks like a normal Java class, apart from the *const_* annotations, but, as described earlier, *CodeGen* (see 5.2.2) will create a remote interface and implementation wrapper for it and *CTLcc* (see 5.2.3) will generate interfaces for using it from other languages.

4.2.3 Implementing the client

Using the component is not much more difficult, the client/service pair shown here assumes that the component service is not running and has to be started by the client (see 4.1.15 for information about retrieval and storage for information available components). The *CTL* package and the package where the remote interface resides (*javaSys* in this case) need to be imported. First, you have to tell the CTL4j where the component service should be started, this is done with a *Location* (see 4.1.6). Then, a *CTL.Process*[9] is created, which takes two arguments, a *Location* object and an integer constant to specify the underlying communication protocol, and passed to the remote interface class that should be used. After that, you can invoke methods of your RI class just as you would do when working locally. When the application is done, it has to terminate the remote process explicitly by using the method *stopService()*. Apart from the initialization and termination, using the CTL components also works just like using local classes.

Listing 25: Calculator client

```
import CTL.Types.*;
import javaSys.*;

public class Client
{
        public static void main (String args[])
        {
                Location loc = new Location("Server", "/path/to
                    /example/simple/", "host", "user", "pass");
                CTL.Process proc = new CTL.Process(loc, CTL.
                    Process.TCP);
                CalcRI.use(proc);
                System.out.println("3␣+␣4␣=␣"+CalcRI.add(3, 4))
                    ;
                System.out.println("5␣-␣4␣=␣"+CalcRI.sub(5, 4))
                    ;
                proc.stopService();
```

[9]The package name has to be specified, because the Java framework already defines a Process class. In subsequent releases, this will called CTL.Link which mimics ctl::link from CTL/C++.

```
        }
}
```

4.2.4 Implementing the server

The source code shown here is a basic service for any components in *$CLASSPATH*, as the CTL4j will automatically look for classes as needed via the Reflection-API. It supports both the daemon-mode, in which it always runs and waits for client requests, and being started by the client via ssh.

Listing 26: Calculator service

```
import CTL.*;
import CTL.Types.*;

public class Server
{
        public static void main (String[] args)
        {
                try
                {
                        boolean dmn = true;
                        int port = 0;

                        if (args.length > 0)
                        {
                                port = RUtil.tryInt(args[0]);
                                dmn = (port != -1);
                        }

                        if (!dmn)
                                Env.grp = new Group(args);
                        else
                                Env.grp = new Group("localhost"
                                    , port, 0, 2,
                                        Location.TCP, true);

                        Env.grp.run();
                }
                catch (Exception e)
                {
                        RUtil.except(e);
                }
        }
}
```

4.2.5 Using the distributed system

CTL4j applications are compiled with Apache Ant. Below, an example *build.xml* is provided for compiling the calculator:

Listing 27: Ant build.xml

```
<project name="example" default="run" basedir=".">
```

```
<description>Simple CTL4j example.</description>

<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="dist" location="dist"/>
<property name="cache" location="depcache"/>

<property name="debug" value="true"/>
<property environment="env"/>
<property name="ctl4j" value="${env.HOME}/projects/wire
    /ctl4j/build"/>
<property name="classpath" value="${env.CLASSPATH}${
    path.separator}${build}${path.separator}${ctl4j}"/>
<property name="example" value="Client"/>

<target name="run" depends="compile" description="Run␣
    example.">
        <java classname="${example}" classpath="${
            classpath}" fork="true">
                <jvmarg value="-Dfile.encoding=ISO
                    -8859-1"/>
        </java>
</target>

<target name="compile" depends="init" description="
    Compile.">
        <depend srcdir="${src}" destdir="${build}"
            cache="${cache}"
                closure="yes"/>
        <javac srcdir="${src}" destdir="${build}"
            nowarn="true"
                debug="${debug}" excludes="Client.java,
                    Server.java"
                classpath="${classpath}"/>
        <java classname="CodeGen.Main" classpath="${
            classpath}" fork="true">
                <arg value="Impl.Calc"/>
        </java>
        <javac srcdir="${src}" destdir="${build}"
            nowarn="true"
                debug="${debug}" classpath="${classpath
                    }"/>
</target>

<target name="init" description="Initialize.">
        <tstamp/>
        <mkdir dir="${build}"/>
        <mkdir dir="${cache}"/>
</target>

<target name="clean" description="Cleanup.">
        <delete dir="${build}" quiet="true"/>
        <delete dir="${dist}" quiet="true"/>
        <delete dir="${cache}" quiet="true"/>
```

```
                <delete file="src/javaSys/CalcRI.java"/>
        </target>
</project>
```

You can use this for most applications. Just keep in mind that the CTL4j classes have to be in the CLASSPATH and that all classes which should form components have to be fed to *CodeGen.Main* as seen in the *compile* rule. See [pro] for more information about writing Ant build-files.

4.3 Example: A distributed Dijkstra algorithm

This section covers a more advanced CTL4j example application, which illustrates the abstraction of binary types and writing user-defined data structures. Readers should already be familiar with the concepts discussed in the last section.

4.3.1 Dijkstra's algorithm

Dijkstra's algorithm, named after its inventor, Dutch computer scientist Edsger Dijkstra, is an algorithm that solves the single-source shortest path problem for a directed graph with nonnegative edge weights ([Wik]).

4.3.2 Serializing a graph

There are several possibilities for shaping a graph data structure, for example associating each node with an array of incident edges. For this example, a graph was chosen to consist of a list of nodes and a matrix which hols an entry at position (i, j) if an edge between i and j exists. For large graphs, this is a quite inefficient data structure in terms of memory footprint. For a distributed application, this means that it is also a waste of bandwidth.
All in all, an ultimate solution for the problem of storing graphs does not exist, application programmers might choose different implementations depending on the task at hand. At this point, a CTL-specific way of doing things comes into play: the abstraction of types from their binary representation. If all communication partners agree on the layout of a graph in the stream, they can talk to each other no matter how they implemented their data structures.
As a graph can have cycles and each node can have multiple edges, deciding on a good serialization is not straightforward. Fortunately, the *Reference* (see *see* reference type provides a means to send larger data structures which contain lots of internal pointers (edges in our case). Therefore, the graph can be written as an array of tuples (i, j), which are pointers to the two nodes of an edge. On the first occurence of a node in such a tuple, the entire data and not only a pointer will be written to the stream. A node itself is also a tuple (String, Integer), storing the name and the weight of the element.

Listing 28: CTL.Types.Node

```
package CTL.Types;

import CTL.*;
import CTL.Streams.*;
import ReflWrap.*;

import java.io.*;
```

```
import java.lang.reflect.*;

/** Element of CTL graph */
public class Node extends Tupel
{
        /** Type parameters of the underlying Tupel */
        private static TypeTree[] types;

        static
        {
                types = new TypeTree[2];
                try
                {
                        types[0] = new TypeTree(String.class);
                        types[1] = new TypeTree(Integer.class);
                } catch (ClassNotFoundException e) {} // Never
                    reached.
        }

        /** This method just sets the statically defined 'types
            ' array
         * to this objects 'type' attribute, as defined in
            Tupel.
         * @param t Dummy argument to satify the TemplHack
            interface used
         *      by Tupel.
         */
        public void setTypes (TypeTree[] t)
        {
                // After 5h of debugging, this line was born,
                    heh.
                this.type = types;
        }

        /** Generate a new Node
         * @param name Name
         * @param cost Cost to reach this node
         */
        public Node (String name, int cost) throws CTLException
            ,
                ClassNotFoundException
        {
                super(types);
                setItem(0, name);
                setItem(1, cost);
        }

        /** Retrieve the name of this Node
         * @return String
         */
        public String name ()
        {
                try
                {
```

```
                return (String)item(0);
        }
        catch (CTLException e)
        {
                return null;
        }
    }

    /** Retrieve the cost of this Node
     * @return Cost
     */
    public int cost ()
    {
        try
        {
                return (Integer)item(1);
        }
        catch (CTLException e)
        {
                return -1;
        }
    }

    /** Retrieve a String representation of this object
     * @return String
     */
    public String toString ()
    {
        return name();
    }

    /** Serial read function
     * @param in Input stream
     */
    public void read (OIStream in) throws IOException,
        ClassNotFoundException,
            InstantiationException, IllegalAccessException,
                InvocationTargetException
    {
        //System.out.println("moongoo");
        try
        {
                setItem(0, in.readString());
                setItem(1, in.readInt());
                //System.out.println(item(0)+" "+item
                    (1));
        }
        catch (CTLException e)
        {
                RUtil.except(e);
        }
    }
}
```

The Node class shows how to write a user-defined data-structure which extends a CTL

type. It is very important that the *TypeTree* is set to the correct values for your type, in this case, a node is a tupel of a string (the name) and an integer (the cost to reach the node). Those need to be passed to the superclass, otherwise the tupel's read method will not know how to read your type. This procedure is part of the template-hack (see 5.2.3) which handles template parameters in the CTL4j. Depending on the type, the user needs to provide his own *read()* and/or *write()* methods specialized for the type. As you can see, node only provides *write()*, because the tupel's default *read()* works fine in this case. The rest of the class is just some code specific to the node and does not matter in terms of CTL4j.

Listing 29: CTL.Types.Tupel

```
package CTL.Types;

import CTL.*;
import CTL.Serialize.*;
import CTL.Streams.*;
import ReflWrap.*;

import java.io.*;
import java.lang.reflect.*;

/** CTL Tupel (a fixed-sized array of multiple types) */
// TODO: variable number of type parameters
public class Tupel<A,B> implements Writable, TemplHack
{
        /** Type paramenters */
        protected TypeTree[] type;
        /** Stored data */
        protected Object[] data;

        /** Array helper function
         * @param array Array of classes
         * @param idx Index number
         * @param moo New value
         * @return Array of classes with the specified value
             replaced
         */
        protected static Class[] insert (Class[] array, int idx
            , Class moo)
        {
                array[idx] = moo;
                return array;
        }

        /** Set the type parameters
         * @param types Array of classes
         */
        public void setTypes (TypeTree[] types)
        {
                //System.out.println("Setting types to: "+
                    Arrays.toString(types));
                type = types;
        }
```

```
/** Retrieve a string representation of this object
 * @return String
 */
public String toString ()
{
        StringBuffer buf = new StringBuffer();
        buf.append("Tupel␣(ID␣#"+hashCode()+"):\n");
        for (int i=0;i<type.length;i++)
                buf.append("\t"+type[i]+":␣"+data[i]+
                        ((i!=type.length-1) ? "\n" : ""
                           ));
        return buf.toString();
}

/** Dummy constructor to make subclasses happy */
protected Tupel ()
{
}

/** Generate a new Tupel
 * @param type Type parameters
 */
public Tupel (Class[] type) throws CTLException,
    ClassNotFoundException
{
        if (type == null)
                throw new CTLException("Invalid␣Tupel."
                    );

        this.type = new TypeTree[type.length];
        this.data = new Object[type.length];
        for (int i=0;i<type.length;i++)
                this.type[i] = new TypeTree(type[i]);
}

/** Constructor from TypeTree */
public Tupel (TypeTree[] tree) throws CTLException,
    ClassNotFoundException
{
        if (tree == null)
                throw new CTLException("Invalid␣Tupel."
                    );

        //System.out.println(java.util.Arrays.toString(
            tree));
        this.type = tree.clone();
        this.data = new Object[tree.length];
}

/** Retrieve the number of elements this Tupel can
    store
 * @return Number of elements
 */
```

```
public int length ()
{
        return data.length;
}

/** Retrieve the type of a specific element
 * @param i Index number
 * @return Type of the element
 */
public Class type (int i) throws CTLException
{
        if ((i<0)||(i>=type.length))
                throw new CTLException(i+"␣is␣out␣of␣
                    bounds.");
        return type[i].getType();
}

/** Retrieve the value of a specific element
 * @param i Index number
 * @return Value of the element
 */
public Object item (int i) throws CTLException
{
        if ((i<0)||(i>=data.length))
                throw new CTLException(i+ "is␣out␣of␣
                    bounds.");
        return data[i];
}

/** Set the value of a specific element
 * @param i Index number
 * @param data New value
 */
public void setItem (int i, Object data) throws
    CTLException
{
        if (data!=null)
        {
                if ((i<0)||(i>=this.type.length))
                        throw new CTLException(i+ "␣is␣
                            out␣of␣bounds.");

                ClassInfo c1 = new ClassInfo(type[i].
                    getType());
                ClassInfo c2 = new ClassInfo(data.
                    getClass());

                if (!c1.equals(c2))
                        throw new CTLException("Type␣
                            mismatch:␣"+c1.name()+
                                "␣!=␣"+c2.name());
        }

        //System.out.println(data);
```

```
        this.data[i] = data;
}

/** Serial read function
 * @param in Input stream
 */
public void read (SerialIn in) throws IOException,
    ClassNotFoundException,
        InstantiationException, IllegalAccessException,
            InvocationTargetException
{
        for (int i=0;i<type.length;i++)
                data[i] = IStream.readType(in, type[i].
                    getType());
}

/** Serial write function
 * @param out Output stream
 */
public void write (SerialOut out) throws IOException,
    IllegalAccessException,
        InvocationTargetException
{
        for (int i=0;i<data.length;i++)
                IStream.writeType(out, data[i]);
}

/** Check if two objects are equal
 * @param t Object to compare to
 * @return True if equal, false otherwise
 */
public boolean equals (Object t)
{
        if (!(t instanceof Tupel) || t==null)
                return false;

        try
        {
                if (length()!=((Tupel)t).length())
                        return false;
                for (int i=0;i<length();i++)
                        if ((item(i) == null && ((Tupel
                            )t).item(i) == null) ||
                                (!item(i).equals(((
                                    Tupel)t).item(i))))
                                return false;
                return true;
        }
        catch (CTLException e)
        {
                Env.log.log_msg(Logger.ERR, "Tupel.
                    equals():␣"+e);
                return false;
        }
```

```
        }
}
```

The Tupel class is used here as an example of how a template class can be written when using CTL4j. Each template class has to implement the *TemplHack* interface, which defines this method:

Listing 30: The ReflWrap.TemplHack interface

```
public interface TemplHack
{
            void setTypes (TypeTree[] types);
}
```

A TypeTree is a tree structure which keeps information about the template parameters of a class available at runtime by using *Class* objects as nodes. As the templates defined by Java's own generics are only available at compile-time, information is passed to objects by constructor arguments for types which are written and by arguments to *serialRead()* for types which are read. The user-defined *setTypes()* method is invoked automatically by CTL4j stream reading code to pass that information to the underlying classes. For appliation programmer this means that he has to provide that method, a field in the class to store the information for his type and a way to pass template arguments to the constructor.

Listing 31: CTL.Types.Graph

```
package CTL.Types;

import CTL.Serialize.*;
import java.io.*;
import java.lang.reflect.*;

/** Unoptimized trivial graph class; show case for CTL.
    Reference
 *
 * Bandwidth usage: 6n + 18m + 9
 *      n: number of nodes; m: number of edges
 *      assumption: each node has at least one edge connected
     to it
 *
 * Sending data and matrix directly: n^2 + 2n + 12
 */
public class Graph implements Writable
{
        private Node nodes[] = null;
        private boolean adjM[][] = null;
        private int edges = 0;

        public Graph (int size)
        {
                if (size<=0)
                        return;
                resize(size);
        }

        public int size ()
```

```
{
        if (nodes == null)
                return 0;
        return nodes.length;
}

public boolean equals (Object moo)
{
        if ((moo instanceof Graph) && toString().equals
            (moo.toString()))
                return true;
        return false;
}

private void resize (int size)
{
        nodes = new Node[size];
        adjM = new boolean[size][size];

        for (int i=0;i<size;i++)
                for (int j=0;j<size;j++)
                        adjM[i][j] = (i==j) ? true :
                            false;
}

public boolean addNode (Node data)
{
        if (nodes==null)
        {
                resize(1);
                setNode(0, data);
                return true;
        }

        if (findNode(data)>-1)
                return false;

        Node old_nodes[] = nodes;
        boolean old_adjM[][] = adjM;
        edges = 0;
        resize(old_nodes.length+1);
        for (int i=0;i<old_nodes.length;i++)
        {
                setNode(i, old_nodes[i]);
                for (int j=0;j<old_nodes.length;j++)
                        if (old_adjM[i][j])
                                addEdge(i, j);
        }
        setNode(old_nodes.length, data);
        return true;
}

public int findNode (Node data)
{
```

```
        if (data!=null && nodes!=null)
        {
                for (int i=0;i<nodes.length;i++)
                        if (data.equals(nodes[i]))
                                return i;
        }
        return -1;
}

public boolean isEdge (int i, int j)
{
        return adjM[i][j];
}

public int cost (int i, int j)
{
        return nodes[i].cost() + nodes[j].cost();
}

public boolean addEdge (int i, int j)
{
        if (i<0 || j<0 || adjM[i][j] || i==j ||
                i>=adjM.length || j>=adjM[i].length)
                return false;
        adjM[i][j] = true;
        edges++;
        return true;
}

public boolean addEdge2 (int i, int j)
{
        return addEdge(i, j) && addEdge(j, i);
}

public void setNode (int i, Node data)
{
        nodes[i] = data;
}

public Node node (int i)
{
        return nodes[i];
}

public String toString ()
{
        if (nodes==null) return "";
        StringBuffer buf = new StringBuffer();
        for (int i=0;i<nodes.length;i++)
        {
                if (nodes[i]==null) continue;
                buf.append(nodes[i]+"␣("+nodes[i].cost
                    ()+")␣-->␣");
                for (int j=0;j<nodes.length;j++)
```

```
                        if ((i!=j) && adjM[i][j])
                                buf.append(nodes[j]+"␣␣
                                    ");
                buf.append("\n");
        }
        return buf.toString();
}

public void write (SerialOut out) throws IOException,
    IllegalAccessException,
        InvocationTargetException, CTLException
{
        out.writeInt(edges);
        if (nodes == null)
                return;

        for (int i=0;i<nodes.length;i++)
                for (int j=0;j<nodes.length;j++)
                        if (i!=j && adjM[i][j])
                        {
                                out.serialWrite(new
                                    Reference<Node>(
                                    nodes[i]));
                                out.serialWrite(new
                                    Reference<Node>(
                                    nodes[j]));
                        }
}

public void read (SerialIn in) throws IOException,
    ClassNotFoundException,
        InstantiationException, IllegalAccessException,
             InvocationTargetException
{
        int len = in.readInt();
        //System.out.println(len);

        for (int i=0;i<len;i++)
        {
                Reference<Node> ref = (Reference<Node>)
                    in.serialRead(
                                Reference.class, Node.
                                    class);
                Reference<Node> ref2 = (Reference<Node
                    >)in.serialRead(
                                Reference.class, Node.
                                    class);
                Node n1 = (Node)ref.obj();
                Node n2 = (Node)ref2.obj();

                //System.out.println(n1+" "+n2);

                if (ref.first()) addNode(n1);
                if (ref2.first()) addNode(n2);
```

```
                        addEdge(findNode(n1), findNode(n2));
                }
        }
}
```

The Graph class shows how to implement a completely new user-defined data-structure, which is quite simple. Each class which wants to be send over a serialized stream has to implement the *Writable* interface, which is similar to the Java-RMI interface *Serializable*. It consists of two required methods, which were already present in the other example types, *read()* and *write()* which define the reading and writing of the type, using the corresponding methods for the fundamental types, like *writeInt()*, as each complex type can be broken down to an aggregation of fundamental types. To minimize the sent data, the graph uses *Reference* objects, which basically are pointers in a serialized stream. Each object gets a unique ID and only the first occurence gets written with all the data, the remaining pointers to the same object write as the unique ID only (see *see* CTL.Types.Referenceor a more accurate description of its inner workings). The rest of the class' code is not CTL4j specific.

4.3.3 The algorithm itself

Listing 32: Dijkstra interface

```
package Impl;

import CTL.Types.*;

import java.util.*;

// See: http://de.wikipedia.org/wiki/Dijkstras_Algorithmus
public class Dijkstra
{
        private Graph g;
        private int[] distance;
        private int[] ancestor;
        private LinkedList<Integer> V;

        public Dijkstra (Graph g)
        {
                this.g = g;
                distance = new int[g.size()];
                ancestor = new int[g.size()];
                V = new LinkedList<Integer>();
        }

        public Graph getGraph ()
        {
                return g;
        }

        // TODO: Arguments need to be called arg[0-9] atm
        public LinkedList<CTL.Types.Node> shortestPath (Node
            arg0, Node arg1)
        //public LinkedList<Node> shortestPath (Node start,
            Node end)
```

```
{
        Node start = arg0;
        Node end = arg1;

        for (int i=0;i<g.size();i++)
        {
                distance[i] = Integer.MAX_VALUE;
                ancestor[i] = -1;
                V.add(i);
        }

        int s = g.findNode(start);
        int z = g.findNode(end);

        if (s==-1)
                throw new RuntimeException("Node␣"+
                    start+"␣not␣found.");
        if (z==-1)
                throw new RuntimeException("Node␣"+end+
                    "␣not␣found.");

        distance[s] = 0;
        ancestor[s] = s;

        while (V.size()>0)
        {
                int u = minDistance();
                V.remove(new Integer(u));

                if (u==z)
                        break;

                for (int i : V)
                        if (g.isEdge(u, i) && distance[
                            u] + g.cost(u, i) < distance
                            [i])
                        {
                                distance[i] = distance[
                                    u] + g.cost(u, i);
                                ancestor[i] = u;
                        }
        }

        LinkedList<Node> ret = new LinkedList<Node>();
        int cur = z;
        while (cur!=s)
        {
                ret.addFirst(g.node(cur));
                cur = ancestor[cur];
        }
        ret.addFirst(start);
        return ret;
}
```

```
        private int minDistance ()
        {
                int ret = V.peek();
                for (int i : V)
                        if (distance[i]<distance[ret])
                                ret = i;
                return ret;
        }
}
```

The implementation of the algorithm is pretty straightforward and follows the original pseudo-code. At the moment, the CTL4j's Reflection parser still has one shortcoming you can see in the *shortestPath()* method: the arguments of method which uses templates have to be called argN, N $\in 0..9$. Apart from this, the programmer of the algorithm code needs no knowledge of CTL4j's API, as you can see.

4.3.4 Implementing the client

Listing 33: DijkstraRI client

```
import java.util.LinkedList;

import CTL.Types.*;
import javaSys.*;

public class Client
{
        private static Node a = null, e = null;

        private static Graph graph ()
        {
                Graph g = new Graph(0);

                try
                {
                        a = new Node("A", 1);
                        e = new Node("E", 1);

                        g.addNode(a);
                        g.addNode(new Node("B", 1));
                        g.addNode(new Node("C", 3));
                        g.addNode(new Node("D", 1));
                        g.addNode(e);

                        g.addEdge(0, 1);
                        g.addEdge(1, 3);
                        g.addEdge(3, 4);
                        g.addEdge(0, 2);
                        g.addEdge(2, 4);
                } catch (Exception ex) {}

                return g;
        }

        public static void main (String args[])
```

```
        {
                LinkedList<Location> locs = Location.parseFile(
                    "locs.txt");
                if (locs.size() != 1)
                        System.exit(1);
                CTL.Process proc = new CTL.Process(locs.get(0))
                    ;
                DijkstraRI.use(proc);
                Graph g = graph();
                DijkstraRI dj = new DijkstraRI(g);
                Graph g2 = dj.getGraph();
                System.out.println("Graph␣check:␣"+g.equals(g2)
                    );
                System.out.println("Shortest␣path:␣"+dj.
                    shortestPath(a, e));
                proc.stopService();
        }
}
```

As always, implementing a client application for an available CTL component is very easy.

4.3.5 Implementing the server

As said before, the service can always be recycled, because loading and finding the components is handled by the JVM itself and by the Reflection API. The basic service is shown verbatim in section 4.2.4.

4.3.6 Using the distributed system

The Ant buildfile shown in section 4.2.5 can also be used again. A sample run of the Dijkstra client looks like this

```
Lenin:~/projects/wire/studienarbeit/examples/dijkstra$ ant
Buildfile: build.xml

init:

compile:

run:
    [java] Graph check: true
[java] Shortest path: [A, B, D, E]
[java] Total execution time 10.408843 seconds.
[java] Clean termination.

BUILD SUCCESSFUL
Total time: 11 seconds
```

5 Method

This section describes the design and implementation of the CTL4j and requires basic understanding of the CTL protocol. If you are not familiar with it, you should read

section 4.1 first.

5.1 Initial design

The initial idea of the Java reimplementation of the CTL/C++ was to avoid writing a parser for the implementation classes (as the C++ implementation did by using templates) and to directly generate the remote interface definition for them. Another goal was learning about the new mechanisms Java 5.0 introduced: Generics (Java version of templates) and Annotations (basically a strict syntax for comments, which can be kept and processed in the compiled bytecode). Avoidance of the parser was accomplished by using the Reflection-API, which is a mechanism to gain information about classes, methods, etc. at runtime and a means to invoke them dynamically.

5.2 Implementation

The Java implementation of the CTL (CTL4j) consists of three major parts:

- ReflWrap: A wrapper around the Reflection-API, which provides an easier interface and some additional convenience features.
- CodeGen: The code generator for remote interfaces (RIs) which utilizes ReflWrap to get the necessary information. In contrast to the other distributed object frameworks (see 2) CodeGen generates RIs from the implementation classes themselves and not from some intermediate language like CORBA's IDL. The generated classes will be named after the implementation classes with the additional prefix 'RI' and will be useable almost completely transparent by applications.
- CTL: This package consists of the core classes needed for remote communication via the CTL protocol. Application programmers only have to know a very small subset of it, the rest will be used by the generated implementation classes only.

5.2.1 ReflWrap

This package provides an easy-to-use interface to the Reflection-API of Java. It also handles possibly allocated static *FunctionIDs* (see 4.1.6) by providing sorted arrays of methods. It consists of the major classes *ClassInfo*, *ConstructInfo* and *MethodInfo*. All the uglyness of wrapper classes and weird notation returned by Reflection calls are hidden by some static methods in the *Refl* class.

TypeTree

The *TypeTree* is a data structure used to pass around information about template parameters, because Java implements generics using *erasure* [10]. The *TypeTree* is, obviously, a tree structure of a template type. An array of arrays of strings would be declared like this:

```
TypeTree tree = new TypeTree("CTL.CCompat.CArray<CTL.CCompat.
    CArray<java.lang.String>>");
```

The *CArray* type is just a C compatible way of declaring arrays in a template-like syntax, instead of using the usual Java notation.

[10] see 5.2.3 for a detailed description.

5.2.2 CodeGen

The code generator generates remote interface definitions and wrapper implementations from standard Java classes by using the Reflection-API via *ReflWrap*. It only generates code if either the implementation or the code generator itself were modified after its last invocation. If possible, it also calls *astyle* ([Dav]), a source code pretty printer, to make the generated code more readable.

5.2.3 CTL

The *CTL* package implements the communication layer of the protocol, the standard data structures, the IDL compiler and several helper classes. It consists of several sub-packages, because of the huge number of classes available (65 at the time of this writing).

CTL.Annotate
Contains annotations for marking elements for special treatment by the code generator.
CTL.CCompat
Contains classes for compatibility with the C++ CTL.

- **CTLcc**

  This class can generate CTL component interface (CI) from a Java class to be able to invoke them from C, C++ and Fortran code. As the code generator, it uses the Reflection-API to get information about available classes. The conversion tool for CTL CIs *ctlcc.py* uses a grammar (see A) to parse the CIs and generates a Java class with declarations for all available methods and functions. This enables Java code to use components which are written in C, C++ or Fortran.

CTL.Comm
Contains the generic Communicator interface, as well as classes which implement it. As of now, only a TCP/IP communicator is available.
CTL.Serialize
Contains classes necessary for data serialization.

- **SerialIn**

  This is an extension of the Java SDK class *DataInputStream* and the CTL replacement for the *ObjectInputStream*. It handles reading serialized data from any *InputStream* by providing specialized functions for the basic data structures (see 4.1.4). On top of the functions for fundamental types, which the *DataInputStream* already provides, it offers the following methods:

  Listing 34: SerialIn methods

```
public String readString ();
public String readWString ();
public Object[] readArray (String type);
public Tupel readTupel (String[] types);
public Object readObject ();
public Object serialRead (TypeTree tree);
```

  The first three methods should be straightforward, but the difference between *readObject()* and *serialRead()* is not as obvious. The former is used to read an

Any object, where the latter reads an object of the type specified in the *TypeTree* data structure.

The *serialRead()* method is used to send arbitrary and possibly parameterized types, which have to implement the *Writable* interface to be written to the stream (see 4.1.7). The first argument is the class of the type, the following are the possible type parameters.

Listing 35: Example of a serialRead() call

```
// Read a LinkedList<String>
in.serialRead(new TypeTree("java.lang.LinkedList<java.lang.
    String>");
```

For passing the information about type parameters to the underlying *read()* call, each parameterized type has to implement the interface *ReflWrap.TemplHack*.

Listing 36: The ReflWrap.TemplHack interface

```
public interface TemplHack
{
        void setTypes (TypeTree[] types);
}
```

This interface is needed, because the Java generics are implemented by type erasure, which means that there is no information available about the type parameters at runtime[11]. This choice was made by the Java developers because of backwards compatibility to the old raw types (see [Gaf04] and [Eck04] for discussion about this). To have this information available for reading types, a class implements this interface and the *serialRead()* method passes it around. The component developer can expect to receive an array of classes before the component's *read()* method is invoked. This interface might be removed from later versions of CTL4j if Sun implements a means to retrieve type information at runtime. See section 4.3 for an example of implementing such a component.

- **SerialOut**

  This is the *OutputStream* analogue to *SerialIn* with the following additional writer functions, on top of *DataOutputStream*:

  Listing 37: SerialOut methods

```
public void writeString (String data);
public void writeWString (String data);
public <T extends Object> void writeArray (T data);
public void writeObject (Object data);
public void serialWrite (Object data);
```

  In contrast to the *SerialIn*, there is no *writeTupel* function, because the type information is already present in the *Tupel* and can therefore be sent with the regular *writeObject()*/*serialWrite()* methods without problems.

CTL.Streams
Contains generic communication stream classes.

[11] It is erased at compile time, hence the name

- **IStream**

  This special stream is handled just like an ObjectOutputStream, however, it saves information about the data which was written to it. Therefore, it is an easy interface to pass around data from different sources to one ObjectOutputStream and it is also capable of writing a 'signature' string to the stream which can be interpreted by the readFromStream() call. This makes it possible to pass around larger amounts of arbitrary data via Object*putStreams w/o recreating the structure of the stream in code on both ends. It is also possible to query the size of the whole stream. Furthermore, the stream can write the data to other streams while keeping the data inside, which makes it ideal for debugging stream related applications.

- **IStream2**

  This stream has mostly replaced *IStream*, because of its simplicity. It stores the serialized data in a byte array, which can be written to the stream directly.

CTL.Types
Contains all CTL specific types, as described in section 4.1.6.
CTL
Some classes, which do not fit into the sub-packages, are located in the top-level CTL package.

- **Group**

  This class is the base for a CTL application. It keeps track of open communication channels, stores the *GroupInfo* objects for all group members and does the initial handshake (see 4.1.8). It can also receive special messages for Group members:

  Listing 38: Special messages

```
@sFID(3) public void recvTermination (PeerID pid, String
    msg);
@sFID(4) public void recvException (String msg);
```

- **Logger**

  The logger writes internal information about events happening inside the CTL code to a file. It can filter messages based on six categories: ERR, WARN, INFO, DBG, DBG1, DBG2.

- **ObjectMap**

  This class handles the mapping from *ObjectIDs* to objects and stores the objects. It uses the *hashCode()* method to generate unique IDs for all objects. Objects can either be created manually and the registered in the *ObjectMap* or a constructor and its arguments are passed to it and the creation happens internally:

  Listing 39: ObjectMap methods

```
public int createObj (Constructor c, Object[] args);
public int regObj (Object obj);
```

  Each *RI* has an internal *ObjectMap* for storing all remotely accessible objects.

5.2.4 Python

In addition to the already available language support (C, C++, Fortran) and the new CTL4j, it is also possible to implement CTL applications, but not components, in Python. This was done using JPype ([Men]), a Python module which allows access to native Java classes. However, the performance of this approach is quite bad. With the help of Swig, Python applications may use C/C++ CTL classes with minor effort.

6 Discussion

6.1 Function Call Overhead/Performance

The measurement was done on a PC-Cluster with 18 Pentium4 (2.4GHz) connected via GigaBit-Ethernet. The following software was used to compile and run the test applications:

CTL	Intel compiler	8.1
CTL4j	Sun JDK	1.5.0_04
.NET remoting	Mono	1.1.8.2
CORBA	ORBit2	2.12.2
Java RMI	Sun JDK	1.5.0_04
SOAP	gSOAP	2.7.3

Listing 40: call a function with this signature

```
array<real8> f(const array<real8>);
```

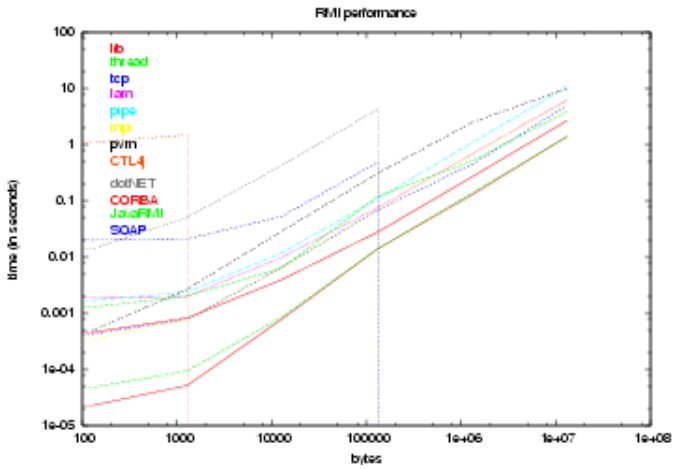


Figure 6: *perf performance*

Table 2: Benchmark results

	0 Kbyte	1 Kbyte	10 Kbyte	100 Kbyte	1 Mbyte	10 Mbyte	100 Mbyte
lib[a]	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$	$2.3 * 10^{-6}$
thread	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$	$1.0 * 10^{-5}$
tcp	$2.5 * 10^{-4}$	$3.7 * 10^{-4}$	$5.2 * 10^{-4}$	$2.9 * 10^{-3}$	$2.9 * 10^{-2}$	$2.8 * 10^{-1}$	$2.8 * 10^{0}$
lam	$2.0 * 10^{-3}$	$1.9 * 10^{-3}$	$2.0 * 10^{-3}$	$1.0 * 10^{-2}$	$7.6 * 10^{-2}$	$6.9 * 10^{-1}$	$6.3 * 10^{0}$
pipe	$3.8 * 10^{-4}$	$6.2 * 10^{-4}$	$2.0 * 10^{-3}$	$1.0 * 10^{-2}$	$8.7 * 10^{-2}$	$8.5 * 10^{-1}$	$8.5 * 10^{1}$
mpi	$3.0 * 10^{-4}$	$3.5 * 10^{-4}$	$7.7 * 10^{-4}$	$6.4 * 10^{-3}$	$7.2 * 10^{-2}$	$7.0 * 10^{-1}$	$5.8 * 10^{0}$
pvm	$4.0 * 10^{-4}$	$5.0 * 10^{-4}$	$2.7 * 10^{-3}$	$2.9 * 10^{-2}$	$3.1 * 10^{-1}$	$2.5 * 10^{0}$	$1.9 * 10^{1}$
CTL4j	$0.3 * 10^{-1}$	$1.1 * 10^{0}$	$1.5 * 10^{0}$	None[b]	None	None	None
.NET	$4.8 * 10^{-2}$	$1.4 * 10^{-2}$	$5.1 * 10^{-2}$	$4.3 * 10^{-1}$	$4.3 * 10^{0}$	None[c]	None[d]
CORBA	$1.7 * 10^{-3}$	$4.6 * 10^{-4}$	$8.3 * 10^{-4}$	$4.0 * 10^{-3}$	$2.8 * 10^{-2}$	$2.7 * 10^{-1}$	$2.7 * 10^{0}$
Java RMI	$1.5 * 10^{-3}$	$1.3 * 10^{-3}$	$2.0 * 10^{-3}$	$6.5 * 10^{-3}$	$1.2 * 10^{-1}$	$5.2 * 10^{-1}$	$3.9 * 10^{0}$
SOAP	$2.1 * 10^{-2}$[e]	$2.0 * 10^{-2}$	$2.1 * 10^{-2}$	$5.3 * 10^{-2}$	$4.9 * 10^{-1}$	None[f]	None

[a] Benchmarks for the different CTL/C++ transports

[b] At the moment, the CTL4j can not send arrays with a size of more than 32k.

[c] Out of memory error while sending. A comparison test on Win32 shows that it takes roughly the same time to send as 1MB.

[d] Out of memory error while sending. A comparison test on Win32 also got an Out of memory error.

[e] One byte was sent, because arrays of 0 byte size cannot be used with gSOAP2.

[f] Example did not compile, because it is not possible to allocate that much space on the stack and using the heap is not possible in this case.

6.1.1 Discussion

As you can see in table, the CTL/C++ and CORBA perform best, with the former being approximately twice as fast as the latter. This has multiple reasons:

- ORBit minimizes *memcpy()* calls by using *readv()* and *writev()*, which makes it very fast for simple arrays. When using the CTL/C++, the performance will be similar if the serialization of the data is similar, no matter if a list, array or tupel is sent.
- ORBit (and CORBA in general) sends data in the hosts byte-order, whereas the CTL/C++ always sends it in network byte-order at this point in time, which means that it needs to swap the byte-order on x86 CPUs.
- In terms of latency, which can be measured when sending 0/1 byte arrays, the CTL/C++ outperforms ORBit by a factor of 6. If the primary use of RMI in your application is message passing, this might be more important than the ability of pushing lots of data around.

You may also notice that CTL4j's performance is one or even multiple orders of magnitude worse than the others. This shows that it is very much work in progress at this point in time and should not be considered for high-performance tasks. Because of this, benchmarks for interoperation with CTL/C++ did not take place, but the CTL4j implements the complete CTL protocol and is able to communicate with components written in other languages. However, the development of it will continue beyond the work on this paper and optimizations will happen at some future date.

6.2 Protocol

When studying the CTL protocol, the reader might wonder why certain design decisions were made. This section explains some of the points which are not immediately clear if you have not designed your own RMI protocol or wrote an implementation of an existing one by yourself. The material in this section is presented as a QA dialog.

- Why do the GroupInfo objects need to be transfered?
  They are used to identify groub members at RMI level independent from the transport protocol.
- Why are IP addresses 128bit and ports 32bit?
  The CTL's IP address data-structure can contain either IPv4 or IPv6 addresses for upwards compatibility of the protocol. The same is true for ports, although no standard defines 32bit wide port numbers, yet.
- Why does the header contain a *PeerID* when the BSD socket interface already provides a lookup for this?
  The CTL protocol was designed to be independent from the transport used. While TCP/IP could work without that information in the header, other protocols, like MPI can not.
- Why do *rPointers* contain a *PeerID*?
  A rPointer can be passed around to other peers which might want to send a message to the object, therefore the rPointer needs to know where the object it points to lives.

6.3 Comparison of distributed object frameworks

In the following sections, the protocolls mentioned ealier (2) will be compared to the CTL in terms of installation, quality and ease of use of APIs, garbage collection for remote objects, serialization, security and the possibility to use them in environments with Network Address Translation (NAT) or firewalls.

6.3.1 CTL/C++

The CTL/C++ consists of headers only and therefore needs no installation apart from copying them to a place were the compiler will find them. It is possible to compile a static or shared library, too, which only needs to be copied to a fitting place. The CTL4j can be easily installed from source.
As described earlier, the CTL requires no knowledge about its inner workings from its users, a component is written just like a normal local class, apart from the CI definition. Through using a resource manager, programming can be done transparently both in terms of location of components and the transport mechanism. Remote and local execution are equal at the source code level. Garbage collection is done by reference counting at the rPointer level.
Support for new transport protocols can be added by writing additional communicators. There is no support for changing data serialization by the user.
Using the pipe transport[12], which tunnels the communication through a SSH connection, authentication, confidentiality and integrity is provided. This transport also allows traversing firewalls and NAT.

[12] As of now, not supported by CTL4j.

6.3.2 CORBA

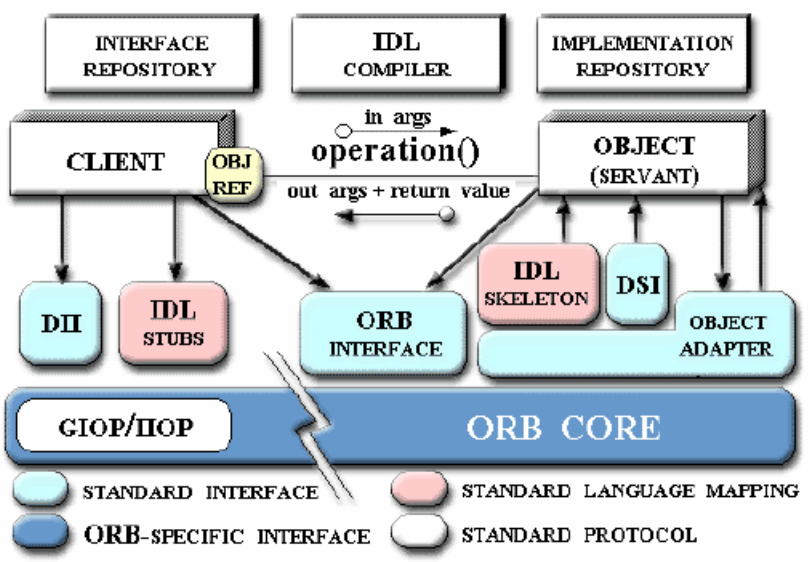


Figure 7: *Shows the architecture of CORBA (source: [Groa])*

ORBit comes with all major Linux distributions, because it is the base of the Gnome desktop environment, but it is also available for other Unix systems such as Solaris os Mac OS X, as well as Windows. The installation size, including dependencies, is about 15MB.
The APIs of CORBA implementations as well as the IDL are designed in a language-independent manor. They define their own fundamental types and composites, which means that the user cannot simply write an implementation without knowing about the communication middleware. Of course, this means that CORBA frameworks can be implemented for any programming language which results in the widest language support of all protocols discussed in this paper.

Listing 41: Example: defining an array of int4

```
typedef sequence<int> int_array;
```

There is no support for general-purpose garbage collection and reference counting is not done on remote objects.
Interoperable Object References (IORs) are CORBAs equivalent to Location objects, as you can see in the example below, they are quite bloated compared to the CTL protocol:

```
Lenin:~/projects/wire/studienarbeit/bench/corba$ ior-decode-2 IOR:010000000c000
Object ID: IDL:Add:1.0
IOP_TAG_GENERIC_IOP: GIOP 1.2[UNIX] pare8:/tmp/orbit-neocool/linc-22e1-0-947f47

IOP_TAG_INTERNET_IOP: GIOP 1.2 pare8:39126
    object_key (28) '0000000083dd67edcaa75080d4a9705ec88fd86b01000000719848e8'
```

```
IOP_TAG_ORBIT_SPECIFIC: usock /tmp/orbit-neocool/linc-22e1-0-947f474dbb5c IPv6
    object_key (28) '0000000083dd67edcaa75080d4a9705ec88fd86b01000000719848e8'

IOP_TAG_MULTIPLE_COMPONENTS:
    IOP_TAG_COMPLETE_OBJECT_KEY: object_key (28) '0000000083dd67edcaa75080d4a97

    Unknown component 0x1
```

The communication between CORBA applications is done via the General Inter-ORB Protocol (GIOP), which is intended as an implementation for the Presentation and Session layers in the OSI network model. For serialization, CORBA uses the Common Data Representation (CDR), a platform-independent formal mapping of data types, as seen in the example below.

```
0x47 0x49 0x4f 0x50 -> GIOP, the key
0x01 0x00           -> GIOP_version
0x00                -> Byte order (big endian)
0x00                -> Message type (Request message)
0x00 0x00 0x00 0x2c -> Message size (44)
0x00 0x00 0x00 0x00 -> Service context
0x00 0x00 0x00 0x01 -> Request ID
0x01                -> Response expected
0x00 0x00 0x00 0x24 -> Object key length in octets (36)
0xab 0xac 0xab 0x31 0x39 0x36 0x31 0x30
0x30 0x35 0x38 0x31 0x36 0x00 0x5f 0x52
0x6f 0x6f 0x74 0x50 0x4f 0x41 0x00 0x00
0xca 0xfe 0xba 0xbe 0x39 0x47 0xc8 0xf8
0x00 0x00 0x00 0x00 -> Object key defined by vendor
0x00 0x00 0x00 0x04 -> Operation name length (4 octets long)
0x61 0x64 0x64 0x00 -> Value of operation name ("add")
0x20                -> Padding bytes to align next value
```

As you can see, the format is similar to the CTL protocol:

- The key corresponds to the CTL protocol's 'magic string'.
- Protocol version.
- Specifying the byte order is not necessary in the CTL.
- Message type and Request ID are similar to the message tag.
- Both protocols carry the message size in their headers.
- Whether or not a response is expected is known through the common interface in the CTL.
- Object keys correspond to Object IDs.
- Operation names correspond to Function IDs.

The main difference are the 'length' fields for any user-defined data and the padding bytes, but basically the message layout of the CTL protocol and the GIOP is quite similar, although the contents are encoded differently.
The CORBA Security Services (CORBAsec) is a part of the core specification since 2.x and defines security functionality interfaces for ORBs, such as encryption via the Secure Inter-ORB Protocol (SecIOP). There are some research projects on adding security to available ORBs, such as [DST].
CORBA 3.x includes a Firewall specification for traversing NAT, but it is not supported by ORBit as of now.

6.3.3 Microsoft .NET

.NET remoting is included in both the Microsoft implementation as well as the open-source alternatives, therefore an installation is not required.
Several languages are supported by the Common Language Runtime (CLR), but the framework itself is only available on the Windows platform. There are two open-source projects, DotGNU and Mono, which aim to produce a compatible alternative compiler and VM. During the research for this paper, Mono was used as platform for developing the examples, because of its better support for remoting.
The API is well integrated into the framework, making it easy to use for developers which are already used to .NET, in addition to that, there is no special syntax to learn for defining interfaces. Garbage colletion for remote objects is provided by .NET remoting.
The .NET remoting architecure is based on five core types of objects (descriptions are taken from [Ram02]):

- Proxies: *These objects masquerade as remote objects and forward calls.* This is done by the rPointers in the CTL.
- Messages: *Message objects contain the necessary data to execute a remote method call.* Data acquired by *Remote.call()*.
- Message sinks: *These objects allow custom processing of messages during a remote invocation.* The functionality is embedded into the CTL at different levels as it does not support different message formats.
- Formatters: *These objects are message sinks as well and will serialize a message to a transfer format like SOAP.* See Message sinks.
- Transport channels: *Message sinks yet again, these objects will transfer the serialized message to a remote process, for example, via HTTP.* The Communicators handle different transport protocols in the CTL. The difference between them and transport channels is, that they can also be used for local message transfer using threads or shared libraries.

For data serialization, .NET uses the SOAP protocol, which is an XML dialect for RMI. In contrast to the CTL, remoting supports different formatters for compatibility with other protocols; for example the Remoting.CORBA project aims to integrate CORBA/IIOP into .NET by defining new formatters and transport channels (see [Joh]). However, this flexibility may be a reason for .NET's weaker performance.

In .NET, Microsoft supports the deployment of application-defined roles and access control based on these roles. These mechanisms are extended in ways that are appropriate for the Internet and heterogeneous environments. Encryption is supported by using HTTPS, in addition to that all other HTTP authentication and authorization mechanisms are provided.
By using HTTP as transport protocol, .NET remoting can easily be used through firewalls and NAT.

6.3.4 Java RMI

Java RMI is part of the Java SDK and therefore no installation is needed.
The API is well integrated into the class framework. Similar to CTL4j, the remote interface is declared as a normal Java interface. There is no location transparency, because every host runs its own naming service. Like all other discussed libraries, Java RMI does not provide an abstraction from local and remote components like the CTL does.
Communication is only possible between Java applications, no bindings for other programming languages are available. As J2EE is not a standard, the conformance varies across vendors, leading to a possible vendor lock-in.
Garbage collection is done on remote objects using the so called Distributed Garbage Collector (DGC).
Data is serialized using a proprietary protocol, called Java Object Serialization, created by Sun. There is no support for using other formatters or transport protocols in Java RMI.
There is no support for authentication or encryption in Java RMI. A Java Specification Request (JSR) ([Mic00]) was rejected for this.
For tunneling through firewalls and NAT, Java RMI supports HTTP as transport protocol instead of the usual raw TCP/IP sockets based communication.

6.3.5 SOAP

gSOAP2 is easily installed from source. Both C and C++ are supported by it, but through using the standard SOAP protocol, communication with nearly any other language is possible.
The APIs of gSOAP2 are integrated into C's type system, but need special treatment for some composite types, such as array, which have to be defined as a struct. However, the framework is similar to the CTL, because it generates all the communication code automatically with a special preprocessor. There is no location independence, because the HTTP server to talk to needs to be defined at source code level.
S̈OAP itself makes no attempt to manage orphaned objects or support remote garbage collection. In fact, the specification explicitly states that this is not addressed by SOAP.̈(source: [KS])
The SOAP protocol is an XML dialect, which leads to a big performance loss, because of the need of parsing the XML document and extracting the data structures from it.
SOAP usually uses HTTP as transport protocol and because of this all usual HTTP mechanisms for security and authentication are supported, like in .NET. The standard itself does not mandate the usage of a specific transport protocol, though.
As .NET, SOAP has no problems with firewalls.

7 Conclusion

The CTL provides an easy-to-use and performant framework for building distributed applications. It offers a uniform behaviour across different remote protocols and local linkage. The protocol itself is significantly easier to understand than related works (the CORBA core specification is ¿1000 pages for example).
In addition to that, during the work for this paper, a new CTL implementation in Java and methods for using it from Python were developed. This means that it can be used for communicating in heterogenous environments were different programming languages are used.

A Component interface grammar

Based on [Nie05].

Listing 42: Component interface grammar

```
<ident> ::= valid C++ identifier
<numargs> ::= Integer # Number of elements
<id> ::= Integer # Unique function ID
<op> ::= overloadable C++ operator

<funcname> ::= <ident> | operator <op>
<type> ::= [const] <ident> [*] | <identifier> "<" <type> ">" |
    "(" <ident> "," "(" <type> [ "," <type> ] ")" "," <numargs>
    ")"
<typelist> ::= "(" ")" [const] "," 0 | "(" <type> [ "," <type>
    ] ")" [const] "," <numargs>
<funcsign> ::= <type> "," <funcname> "," <typelist>

<constructor> ::= #define CTL_Constructor <id> <typelist>
        [ #define CTL_Constructor <id> Throws <typelist> ]
<method> ::= #define CTL_Method <id> <funcsign>
        [ #define CTL_Method <id> Throws <typelist> ]
<smethod> ::= #define CTL_StaticMethod <id> <funcsign>
        [ #define CTL_StaticMethod <id> Throws <typelist> ]
<function> ::= #define CTL_Function <id> <funcsign>
        [ #define CTL_Function <id> Throws <typelist> ]
<functiontmpl> ::= #define CTL_FunctionTmpl <id> <funcsign> <
    typelist>
        [ #define CTL_FunctionTmpl <id> Throws <typelist> ]

<classentry> ::= <method> | <smethod> | <constructor>
<classbody> ::= #include CTL_ClassBegin
<classentry> [ <classentry> ] #include CTL_ClassEnd
<class> ::= #define CTL_Class <ident> <classbody>
<classtmpl> ::= #define CTL_ClassTmpl <ident> ","
<typelist> <classbody>

<libentry> ::= <class> | <function> | <functiontmpl>
<libbody> ::=   #include CTL_LibBegin <libentry>
[ <libentry> ] #include CTL_LibEnd
<library> ::=   #define CTL_Library <ident> <librarybody>

<interface> ::= <library> | <class>
```

B Obtaining the CTL

- CTL
  (...)
- CTL4j
  Daily snapshots in both source and binary form are available at *https://rvgds.dyndns.org/ neocool/ctl4j/snapshots/*. Also available is the daily regenerated JavaDoc documentation at *https://rvgds.dyndns.org/ neocool/ctl4j/doc/*.
- py2c
  The experimental py2c code (without documentation as of now) is available at *https://shuya.ath.cx/ neocool/code/py2c/*.
- Resource manager
  (...)
- ri-generator
  (...)

References

[Dav] Tal Davidson. Artistic Style homepage. http://sourceforge.net/projects/astyle/.

[DST] DSTC. CORBA security research homepage. http://security.dstc.edu.au/projects/corba/.

[Eck04] Bruce Eckel. Puzzling Through Erasure II. http://mindview.net/WebLog/log-0058, April 2004.

[For] The Common Component Architecture Forum. Homepage of the CCA Forum. http://www.cca-forum.org/.

[Gaf04] Neal Gafter. Puzzling Through Erasure: answer section. http://gafter.blogspot.com/2004_09_01_gafter_archive.html, September 2004.

[Groa] Object Management Group. ORB architecture. http://www.dalmatian.com/CORBA/figure2.gif.

[Grob] W3C XML Protocol Working Group. SOAP Specifications. http://www.w3.org/TR/soap.

[Inca] UserLand Software Inc. XML-RPC homepage. http://www.xmlrpc.com/.

[Incb] ZeroC Inc. Ice homepage. http://www.zeroc.com/.

[JCr] JCraft.com. Java Secure Channel homepage. http://www.jcraft.com/jsch/.

[Joh] Kristopher Johnson. Remoting.CORBA homepage. http://remoting-corba.sourceforge.net/.

[KS] Mark C. Stiver Kennard Scribner. Understanding the Simple Object Access Protocol. http://www.developer.com/xml/article.php/641321.

[McG] Paul McGuire. PyParsing homepage. http://pyparsing.sourceforge.net/.

[Men] Steve Menard. JPype homepage. http://jpype.sourceforge.net/.

[Mic] Microsoft. Component Object Model Technologies. http://www.microsoft.com/com/.

[Mic00] Sun Microsystems. Java Specification Request 76: A high-level API for network security. http://www.jcp.org/en/jsr/detail?id=76, August 2000.

[MW05] Stephan Stapel; Dr. Stefan Mueller-Wilken. Eiskalt serviert: Verteilte Internet-Anwendungen mit Ice. *ix*, 7:119–123, 2005.

[Nie05] Dr. Rainer Niekamp. CTL Manual for Linux/Unix, 2005.

[pro] Ant project. Apache Ant homepage. http://ant.apache.org/.

[Ram02] Ingo Rammer. *Advanced .NET Remoting (C# Edition)*. Apress, 2002.

[vE] Robert van Engelen. gsoap2 homepage. http://sourceforge.net/projects/gsoap2/.

[Wik] Wikipedia. Dijkstra's algorithm. http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

Index

Basic structure, 8

Comparison with other frameworks, 41
CTL-specific types, 6
CTL.Annotate, 36
CTL.Annotate.any, 6
CTL.Annotate.const_, 12
CTL.CCompat, 36
CTL.CCompat.CTLcc, 36
CTL.Comm, 36
CTL.Group, 38
CTL.Logger, 38
CTL.ObjectMap, 38
CTL.Remote, 10, 11
CTL.RI, 36
CTL.rResult, 7
CTL.Serialize, 36
CTL.Serialize.SerialIn, 36
CTL.Serialize.SerialOut, 37
CTL.Serialize.Writable, 7
CTL.SSHv2, 4
CTL.Streams, 37
CTL.Streams.IStream, 38
CTL.Streams.IStream2, 38
CTL.Types, 38
CTL.Types.FID, 6
CTL.Types.GroupInfo, 6
CTL.Types.Header, 9
CTL.Types.IPaddr, 6
CTL.Types.Location, 7
CTL.Types.PeerID, 7
CTL.Types.rPointer, 7

DAT messages, 10

Exceptions, 12

Goals, 4

Handshake, 8

Implementing a distributed application, 20
Installation, 16
Introduction to RMI, 1

Layers, 8
Limitations, 4

ObjectID, 7

Package CodeGen, 36
Package CTL, 36
Package ReflWrap, 35

Remote answer, 11
Remote call, 10
Remote interfaces, 12
Resource manager, 16

Termination, 8
Terminology, 5
TypeTree, 35

Usage, 16
User-defined types, 7