The Component Template Library Protocol and its Java Implementation

Boris Bügling

Institute for Scientific Computing, Technical University Braunschweig

30. März 2006

Outline

Why Distributed Computing?

- Using the computing power of available workstations efficiently
- Sharing special hardware and software
- Using many smaller machines is often cheaper
- Increased reliability and availability
- Services can be colocated at the provider instead of the user

History

Basic approach

'Manual' serialization, I/O via BSD sockets with read(2) and write(2)

Remote Procedure Calls (RPC)

- Procedural interface for services on remote machines
- Location of services should be transparent
- Serialization using a well-defined protocol → platform- und language-independent systems

History

Remote Method Invocation (RMI)

Invocation of methods of remote objects → Object-oriented RPC

Distributed components

Being independent of

- Location
- Language
- Deployment
- Implementation

How to compare the solutions

- Installation → Shipped with the framework or common Linux distributions
- Quality and ease of use of APIs
- Garbage collection (GC)
- Extensibility (new transports, serializers)
- Security (authentication, integrity, secrecy)
- Firewall and NAT support → Supported by all solutions
- Performance (detailled tables and sample code in the paper itself)

CORBA

- Common Object Request Broker Architecture, standardized by the Object Management Group (OMG)
- Implementation reviewed: ORBit2 by the GNOME project
- Transport-Protocol IIOP, layered on top of TCP
- IDL defines the component interface in a language-independent way
- API: Language independent, but CORBA-specific
- No GC; not easy to extend
- Security: Still under research and not provided by all implementations

Java RMI

- Part of the Java SDK by Sun
- Well integrated into Java; uses Java interfaces → Not language-independent
- GC: Done by the so called Distributed Garbage Collector
- Extensibility: Not possible without modifying Sun's code
- No support for authentication or encryption

Microsoft .NET

- Transport protocol: SOAP (XML) or the binary MS-specific Remoting
- Well integrated into C#, interfaces are defined implicitly by the implementation → Applications need to run inside the .NET runtime
- GC: Supported
- Pluggable formaters and transports, example: Remoting.CORBA
- Role-based access control; encryption via HTTPS

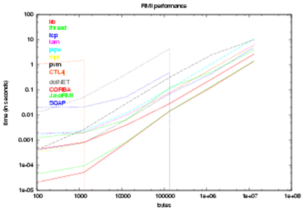
SOAP

- Simple Object Access Protocol, W3C recommendation
- Implementation reviewed: gSOAP2
- XML-based, HTTP usually used as transport protocol
- API: Language- and SOAP-specific
- No GC; not easy to extend
- Supports HTTPS and HTTP-Auth

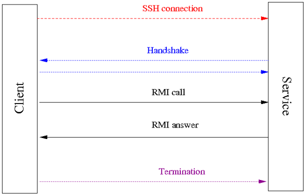
CTL

- API: Applications and components can be developed without knowledge about the CTL itself, well integrated into the supported languages
- GC: Automatic (reference counting at rPointer-level)
- Extensibility: User-defined communicators can provide new transports, no support for new serializers
- Security: Provided by the pipe transport (using SSH)

Performance



Basic structure of communication

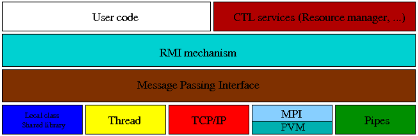


Breaking down complex types

- Fundamentals: integer types, floating point types, void
- Composites
  - Arrays (serialized as: size, e0, e1, ...)
  - String (serialized as: e0, e1, ..., 0)
  - Tuple (fixed size; serialized as: e0, e1, ...)
  - Reference (serialized as: typeID, true, data or typeID, false)
- All other types can be serialized as an aggregate of these types
- User-defined components do not have to deal with the binary data directly
- Protocol implementations just need to understand the binary stream → language independence

Separation of communication path and application

- Layers of the CTL protocol



Separation of interface and implementation

- Example:
  #define CTL_Class AddCl
  #include CTL_ClassBegin
  #define CTL_Method1 int4, add (const int4, const int4), 2
  #include CTL_ClassEnd
- → The implementation just needs to export this interface; multiple (different) solutions possible

Location independence

- CTL.Location
  - name - Component's name
  - host - Hostname
  - path - Filesystem location
  - exec - Executable name
  - link - Transport protocol
- CTL allows resource managers at user-level (using the CTL_Locator interface)
- Mapping: Component's name → location
- Future: User can specify additional requirements (Component Query Language, CQL)

Goals

- Development of applications and components in Java possible ✓
- Documentation of the protocol ✓
- Help to improve the CTL/C++ ✓

Source code comparison of CTL4j and Java RMI

Java RMI

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloClient
{
public static void main (String[] args)
{
try
{
Hello obj = (Hello)Naming.
lookup("//localhost/HelloServer");
String message = obj.sayHello();
System.out.println("RMI msg: "+message);
}
catch (Exception e)
{
System.out.println("HelloClient exception: " + e.getMessage());
e.printStackTrace();
}
}
}
```

CTL4j

```
import javaSys.HelloCI;

public class HelloClient
{
public static void main (String args[])
{
HelloCI obj = HelloCI.create();
System.out.println(obj.sayHello());
}
}
```


Java implementation

- No preprocessor → code-generator required
- The *Reflection* API provides introspection of classes, therefore no Java parser was needed
- Heavy use of Java 1.5 features (Generics, Annotations)
- **Generics** are used to emulate the template syntax of the CTL/C++
  - Template parameters are deleted from the actual compiled class (Erasure) → information is kept in a separate data structure, the *TypeTree*
  - For interfaces, the template parameters need to be present, but cannot be queried properly using *Reflection* → The ByteCode Engineering Library (BCEL) was used to parse the bytecode and provide this information
  - Both problems are handled transparently by the *ReflWrap* package

Java implementation

- **Annotations** were used to implement missing syntax features (*const* modifier, static function IDs)
- Interoperability with the CTL/C++ interfaces is achieved by *ctlcc.py*, a Python script which generates a Java dummy implementation class, which can be fed to the code-generator as usual

Future

- *CTL.Process* → *CTL.Link*
- New transports
  - local calls (implemented in the newest snapshot)
  - Threads
  - Pipe → Security provided by SSH
  - C++ using the Java Native Interface (JNI)
- HTTP as transport protocol (maybe XML serialization) → Webservices
- Optimization (`'First make it work, then make it fast.'`)

Reasons for using the CTL

- Performance: low latency and good scalability
- Uniform behaviour across different transport protocols and local linkage
- Easy to understand protocol (compared to the 1000 pages of the CORBA core specification, for example)
- Portability: C, C++, Fortran, Java and Python are supported languages
- Almost no learning curve for implementing components and applications

Discussion

Any comments or questions?