MVCI – Evolutionary, Dynamically Updatable Externally Multi-Versional Component Framework

Joonas Haapsaari

University of Tampere
Department of Computer Science
Master's Thesis
May 2008
With the proliferation of the software-as-a-service application model and other distributed computing models, ensuring the compatibility of the different pieces of the distributed solutions becomes a complicated task. This is further highlighted by the requirement for the availability of the solutions even during and after a dynamic update of the individual components within the distributed component ecosystem.

This thesis introduces the problem consisting of clients concurrently requiring different versions of the same server components within a component-based ecosystem. In the beginning, the solution domains for the problem are identified and the goals for the solution are laid out. A framework that solves the problem – MVCI – is then introduced. It runs a single version of a server component implementation and allows a number of clients to concurrently use multiple, mutually-incompatible versions of the interfaces of the server component. The framework provides automatic translation from the interface versions not directly supported by the implementation to the versions that are supported by the component implementation. Finally, a reference implementation of MVCI supporting automatic transitive translation of interface versions is described in detail. The reference implementation is a Java-based framework that meets most of the goals laid out in this thesis.

In conclusion, the MVCI framework supports independent evolution of components and provides them the capability for dynamic updates. The framework meets well the goals set in the beginning and the reference implementation of MVCI proves that it is feasible to implement such a system.

Key words and terms: dynamic update, installation, component, component framework, software evolution, interface version, interface translation, transitive translation.
Acknowledgements

It took me a bit more than five years (in calendar time) to complete this thesis. Five years is a long time and I want to thank my wife Mari for her unbelievable patience and my son Eliel for, well, just being there. Next time I'll choose a subject that is more closely related to my job (or a job that is more closely related to the subject, whichever is more convenient). Finally, I would like to thank Jyrki Nummenmaa who acted as my supervisor. His guidance was essential to this thesis, especially in the very early phase and in the final fine-tuning phase.

Tampere, May 18th, 2008

Joonas Haapsaari
Table of Contents

1. Introduction .......................................................................................................................... 1
   1.1. Software components ................................................................................................. 2
   1.2. Component vision ......................................................................................................... 2
   1.3. Definition of software components ............................................................................ 3
   1.4. Component Interfaces .................................................................................................. 3

2. Dynamic change management ................................................................................................. 5
   2.1. Terminology for dynamic updates ................................................................................ 5

3. Running multiple versions of component interfaces concurrently ..................................... 7
   3.1. Environment ................................................................................................................ 7
   3.2. Problem statement ....................................................................................................... 8
      3.2.1. What is compatibility? ........................................................................................ 9
   3.3. Five solution domains for the independent evolution problem .................................. 10
      3.3.1. Application-external domains ............................................................................ 11
      3.3.2. Client domain .................................................................................................... 11
      3.3.3. Middleware domain ......................................................................................... 12
      3.3.4. Server domain .................................................................................................. 12
   3.4. Goals .......................................................................................................................... 13
      3.4.1. Dynamic update of the component implementation ............................................ 13
      3.4.2. Dynamic update of the whole component ............................................................ 14
      3.4.3. No modifications needed to the client components or systems.......................... 14
      3.4.4. State transfer support ....................................................................................... 14
      3.4.5. Multiple versions of interfaces concurrently used by the clients ....................... 14
      3.4.6. Single running implementation serving several interface versions .................... 15
      3.4.7. No constraints on modifying the interface .......................................................... 15
      3.4.8. No constraints on data types .............................................................................. 15
      3.4.9. System should not make development more complicated .................................... 15
      3.4.10. Performance must not degrade ......................................................................... 15
      3.4.11. Programming language and operating system independent ............................... 16

4. MVCI framework ..................................................................................................................... 17
   4.1. MVCI terminology ....................................................................................................... 17
   4.2. MVCI Components ...................................................................................................... 18
      4.2.1. Component interface ......................................................................................... 19
      4.2.2. Evolution of the component interface .................................................................. 20
      4.2.3. Interface translation layer .................................................................................. 21
      4.2.4. Component implementation ............................................................................... 23
      4.2.5. Packaging metadata ......................................................................................... 24
Appendix A: Sample JAR manifest file for the MVCI reference implementation
Appendix B: Source code for two versions of a component interface, an adapter and a translator
Appendix C: MVCI reference implementation performance benchmarks
Appendix D: MVCI source code licensing terms
Appendix E: GNU General Public License, version 2
Appendix F: MVCI reference implementation quick guide
Appendix G: MVCI reference implementation source code in base64 encoded tar.bz2 file
1. Introduction

In the world of electronic commerce, online banking and contract manufacturing the trades are more and more relying on computer-based systems for information exchange and storage. Traditionally banks, insurance companies and other large institutions have utilized custom-made back-end storage server and computing power, business logic, for strategic operations such as deposits and withdrawals in the banking world. Clients have been “dumb” or thin clients that merely allow the teller to execute commands on the back-end business logic mainframe. The actual applications have been running on single mainframe computer.

The world has gone a long way from those days and nowadays it is more and more important for enterprises to have systems that can interact with each others. A good example of this is a field force automation (FFA) solution. According to Wikipedia [2008a], field service management, also known as field force automation, is an attempt to optimize processes and information needed by companies who send technicians or staff "into the field" (or out of the office.) It most commonly refers to companies who need to manage installs, service or repairs of systems or equipment [Wikipedia, 2008a]. The FFA solutions need to integrate to several computer systems, some of which may be hosted by other companies, forming large distributed systems.

The FFA solution in Figure 1 has connections to a customer relationship management (CRM) system, a map- and a navigation provider and an in-house warehouse database. The application gets customer data, such as the contact details, from the CRM and based on that, uses the navigation provider to calculate a route from the current location of the serviceman to the customer's premises. In addition to that, the FFA application fetches the warehouse status data from the warehouse database in order to make sure that the necessary repair parts are available.

![FFA Application Diagram](image)

**Figure 1:** Field force automation application using other solutions in a distributed set-up.
In the FFA solution of Figure 1, only the FFA application and the warehouse database are hosted by the company operating the application. The CRM and the navigation providers are hosted by separate companies and provided as a service to the FFA solution. This means that the company controlling the FFA application does not control certain parts of the whole solution – they are owned by different entities and thus they may be developed in a different cycle.

1.1. Software components

The solution proposed for the problem of large distributed systems is to use *software components*. The CORBA Component Model [CORBA Components, 2000] and the Enterprise Java Beans [EJB 2.0 Specification, 2001] are well known models designed to address some of the key problems of large distributed systems by using a well-defined component model.

The basic idea behind software components comes from other engineering areas where the components are standard building blocks for almost anything imaginable. Szyperski [1998] states that “the use of components is a law of nature in any mature engineering discipline.” Software components are the basic building blocks of most any software and they have been compared to Lego blocks although this comparison is not fair as there are obvious differences [Szyperski, 1998]. According to Szyperski [1998], software is different from other products because it is actually a meta-product. Computers can be seen as fully automated factories and software is the blueprint or plan of the product produced by the computer. Utilization of components moves software one step closer to the Lego world.

1.2. Component vision

Components are units of reuse that provide a ready-made solution to a specific problem. The ultimate vision is that anyone or any company could acquire off-the-shelf software components and combine them in order to get the software product they need. Ideally, it should go much like building something out of Lego blocks but at least currently there usually is a need to write some pieces of software that glue the components together.

The other problem is that in order to happen, the component vision needs a critical mass of components [Szyperski, 1998]. There is little point using general components as a basis of a software product if only a small part of the software can be created using ready-made components. As Szyperski [1998] points out, the components need to be more generic than customized, non-component software and it is much easier to make specific proprietary software than generic. One of the issues hindering the proliferation
of components is the fact that very few component infrastructures proposed so far address the component versioning problem [Szyperski, 1998]. Szyperski [1998] refers to the problem where client components are using services of a server component. There is a clear conflict if a client component requires version 1 of the server component and another client component requires version 2 of the server component – this conflict needs to be addressed by the component infrastructure.

1.3. Definition of software components

There are multiple definitions of software components. Szyperski [1998] says that “software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.” Another definition by Szyperski states:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [Szyperski, 1998]

According to Orfali and Harkey [1998], all distributed objects are components by definition. A distributed object infrastructure can be seen as a component infrastructure that has clearly defined interfaces and components that implement the interfaces, and other components that use those interfaces. Orfali and Harkey further clarify that “components are smart pieces of software that can play in different networks, operating systems, and tool palettes. A component is an object that's not bound to a particular program or application.” [Orfali and Harkey, 1998]. From the definitions of component we can recap that components are self-contained pieces of software that are not dependent on any particular application and that communicate using interfaces.

1.4. Component Interfaces

Interfaces can be seen as contracts between the client components and the server components. The contract states the responsibilities of the server and of the client. The server needs to implement the interface and the client must use the server component in the way defined in the interface. [Szyperski, 1998]

In component software, all services provided by a server component are provided through an interface to the client component. The definition of the interface depends on the component infrastructure in use. For example, in CORBA the interfaces are defined in a special interface definition language, IDL [CORBA, 2002] and in Enterprise Java
Beans the interfaces are defined in Java classes and interfaces [EJB 2.0 Specification, 2001][Joy et al., 2000].

As the only way for a client to access the services of a server component is via the interface of the server component, it means that there is a dependency from the client to the server component's interface. Over the time at least some of the server components need to be developed further and in many cases the interface needs to be modified. This breaks the contract with the client if the component infrastructure does not provide any support for server component evolution.
2. **Dynamic change management**

As the distributed systems evolve, a need for somehow modifying parts of the system usually rises at some point. It has become more and more common that these modifications should occur without interruptions in service – the system must be running even as it is modified. These modifications include upgrading nodes, downgrading nodes, adding new nodes and removing old nodes.

In Figure 1, we introduced an imaginary field force automation application that uses the Google Maps service and the Salesforce.com CRM service. The Google Maps- and the Salesforce.com CRM service are hosted by separate companies using a software-as-a-service model [SIIA, 2000], which means that they need to be able to evolve independently of the field force automation application. In addition to that, they need to be available at all times for applications like the example field force automation solution which means that it is not an option to stop and restart the services when they are updated. The capability for dynamic change management – or a dynamic update – is essential.

Frieder and Segal [1991] define *dynamic update* as the ability to dynamically update a program, i.e., load a new version of a program without stopping the currently running version. According to Hicks *et al.* [2001], a system is *dynamically updatable* if it may be altered while it is running.

Kramer and Magee [1990] describe a model for dynamic change management, which addresses the evolutionary change of the software. The evolutionary changes are the kind of changes that are not anticipated at the initial design time and they are applied as the application is already running [Kramer and Magee, 1990]. Dynamic change management in turn means that it should be possible to apply the evolutionary changes to a part of a system, so that the processing is not interrupted in the part that is not affected by the changes [Kramer and Magee, 1990].

2.1. **Terminology for dynamic updates**

The terminology for component versioning is discussed by Cook and Dage [1999]. They suggest that the terminology should be analogous to the one used in the field of configuration management as it already has terms established. Additionally, Cook and Dage [1999] propose a new term, *fusion*, which has no counterpart in the configuration management field (see Table 1).
Table 1. Component versioning terms (adapted from Cook and Dage [1999])

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>Any unique instance of a component.</td>
</tr>
<tr>
<td>Baseline version</td>
<td>Stable and foundational version of a component.</td>
</tr>
<tr>
<td>Revision</td>
<td>A version of a component that has been modified in some way.</td>
</tr>
<tr>
<td>Variants</td>
<td>Independent descendants of a parent version. Each sibling fixes a single problem independently of the other descendants.</td>
</tr>
<tr>
<td>Fusion</td>
<td>A version that is generated by merging two or more variants. The fusion version has more than one parent version.</td>
</tr>
</tbody>
</table>

The term *version* applies to any unique instance of component. A *baseline* version is a version that proves stable and foundational. A new *revision* is a version modified in some way resulting a linear relationship between the parent version and the revision. If a component version has multiple descendants where each descendant fixes a single problem independent of the others, the descendants are called *variants* forming a tree of versions. When these variants are merged into a single new version it is called a *fusion*. [Cook and Dage, 1999]
3. Running multiple versions of component interfaces concurrently

This chapter contains the problem statement we are assessing in this thesis. In addition, the goals of a multi-versional system are laid out in the end of the chapter.

3.1. Environment

The environment assumed in this thesis is a multi-tier environment where there are components in the role of both client and server. Figure 2 depicts the multi-tiered heterogeneous operating environment of the application server systems. We will concentrate on the application server in the middle and especially on the components in a server component role there. A prime example of such a component is the Component 2 in Figure 2.

A server component may have several concurrent clients from external systems, the same application server environment or even some crossing organizational boundaries. Furthermore, the server component itself may be a client to another component.

![Figure 2. Component 2 has multiple clients (Component 1, 3 and 4) in different environments. Component 2 itself is a client to a remote Component 5.](image)

In Figure 2, there are two components (Component 2 and 5) in server role and four components in client role (Component 1, 2, 3 and 4). The connections between the components (a, b, c and d) depict the client-server component relation. The arrow points to the server component for the relation in question. In Figure 2, it is notable that Component 2 is in dual-role: it is the server component for Component 1, 3 and 4 and a client for Component 5.

There is also an organizational boundary visible in Figure 2. This is an important thing to notice, as the control of the evolution of different components is not in the hands of a
single organization. This highlights the possibility that each component lives according to their own life cycle without the necessity to follow the evolution of other components even if they need to communicate with each other.

Traditionally, in the similar distributed environments as depicted in Figure 2, the responsibility for the compatibility of a client and a server in an upgrade situation falls to both server- and client vendor. This is problematic with the organizational boundary as potentially also the party whose environment has not changed needs to make changes due to the other party. In a perfect world, the responsibility would only fall to the organization making the changes and even in there, to the owner of the particular component.

3.2. Problem statement

The problem this thesis addresses can be seen in Figure 3. There are several client components trying to access the same server component and the clients require different versions of the server component. Typically, only the clients that require exactly the version of the server component installed can access it and the others are left without service. The situation comes up easily if the clients and the server are developed independently of each other, which often is the case in large companies: different parts of the IT subsystems are sourced from different vendors.

In Figure 3, the Client v1 could be developed by an integrator that has since gone out of business – thus preventing rehiring that same integrator to port the client to the new server back-end. On the other hand, the Client v3 could be an internally developed client using the new server back-end (for which the modifications in the back-end were needed for to begin with).

![Figure 3. The incompatible version problem.](image)

It would be an unnecessary cost for the company if the Client v1 could not communicate with the server without modifications. Of course, one could argue that the
The server should have been backwards-compatible in the first place and thus the Client v1 should run without any modifications but this brings another problem: the hands of the developers of the server should not be tied by the (wrong) decisions made in previous versions.

There are at least two solutions to the problem in Figure 3. The easier and the most used solution is to avoid making such changes to the server that would break the compatibility of the old versions. The other and more complex solution is to have such an infrastructure in place that it allows independent evolution of the server by supporting component modifications in the server without the need to worry about the compatibility of the clients. The infrastructure takes care of the compatibility.

3.2.1. **What is compatibility?**

By compatibility of a client component and a server component, we mean that the server component can respond to the client's requests and the client component can interpret those responses. Compatibility is about mutual understanding of the client and the server component.

In a component-based system, compatibility is about the interface between the client component and the server component. The client component uses a specific variant – or version – \( I_{\text{client}} \) of an interface defining the contract between the client and the server. The server component in turn implements a specific version \( I_{\text{server}} \) of the interface. Now, in order for things to work between the client and the server component, the server should generally implement the same version of the interface than the client component uses (so that \( I_{\text{client}} = I_{\text{server}} \)). It is not strictly mandatory for the both parties to have exactly the same version – this depends on the programming language in use. For example in Java, things will work if the server implements a binary compatible superset of the interface the client is using. The Java binary compatibility is defined by Joy et al. [2000] to support the following modifications in the new version of the class or interface:

- Re-implementing existing methods, constructors, and initializers to improve performance.
- Changing methods or constructors to return values on inputs for which they previously either threw exceptions that normally should not occur or failed by going into an infinite loop or causing a deadlock.
- Adding new fields, methods, or constructors to an existing class or interface.
- Deleting private fields, methods, or constructors of a class.
- When an entire package is updated, deleting default (package-only) access fields,
methods, or constructors of classes and interfaces in the package.

- Reordering the fields, methods, or constructors in an existing type declaration.
- Moving a method upward in the class hierarchy.
- Reordering the list of direct superinterfaces of a class or interface.
- Inserting new class or interface types in the type hierarchy.

Different rules apply in different programming languages and environments. For example the rules for C++ depend on how the compiler works for the target environment and these guidelines cannot be directly used.

For this thesis, we take the strict interpretation and assume that a server component and a client component are compatible only if they use exactly the same version of the interface (i.e. $I_{\text{server}} = I_{\text{client}}$). We claim that with the framework presented in chapter 4, there is no need to think about interface binary compatibility other than using the exactly same version of the interface in both ends.

### 3.3. Five solution domains for the independent evolution problem

The problem being addressed by this thesis consists of a system that has several client components and server components where the real challenge is to make the system available during independent evolution of all of the client- and server components. Figure 4 shows all of the domains in which the solution could be implemented.

![Figure 4. Possible domains for implementing the solution for the independent evolution problem: client-side external (a), client (b), middleware (c), server (d) and server-side external (e).](image-url)

There are five approaches to the independent evolution problem, two application-external domains and three application-internal domains. In Figure 4, (a) and (e) are application-external domains, and (b), (c), and (d) are application-internal domains. The difference between domains is further discussed below.
3.3.1. **Application-external domains**

In Figure 4, the Client-side external (a) and Server-side external (e) solutions are external to the application. This means that the application has little control over them, especially during application development. Furthermore, application-external domains are usually controlled by a party that is different from the one controlling the application-internal domains.

An example of a client-side external solution (option (a) in Figure 4) would be making the end user use two different applications, the old one for accessing the old data and a new one for accessing the new data. Any data migration would be done by the end user by the means of manually copying values from one application to another. The problem of this approach is that it rarely works if the system is complex and involves a large amount of data that needs to be migrated, or if the application is used by other applications (i.e. computers, not humans), in which case it may not be feasible to implement the necessary changes to these applications.

The server-side external solution domain (option (e) in Figure 4) ranges from making changes to the hardware to modifying the operating system to changing a software component that is not a part of the application itself. The application's data storage system can be considered to be a part of either the application-internal domain or the application-external domain, depending on the application. As an example, one could potentially solve the version problem with an application-external database that would allow access to two different component versions running in parallel and providing a view of the same data to both of the versions. The problem with this approach is that the business logic usually resides in the component so the database cannot update the logic-part unless the logic is somehow stored to the database as well but in that case one could argue that it no longer is an application-external solution as most of the application is in the database.

3.3.2. **Client domain**

Solving the versioning problem in the client domain (option (b) in Figure 4) involves changing all the clients simultaneously with the server migration so that they always use a single version of any component. This is generally how web browsers relate to the web server – the server provides the content for all web browsers connected to it and the content is updated when the server is updated.

While this solution is working exceptionally well in web-environment, it is not very well suited for a heterogeneous environment involving machine-to-machine communications as the updated interface - web page in this case – needs to be
interpreted correctly, which is not an easy task for computers. In general, there is always a need to manually update each client component – at least to integrate the modified interface to the client software that accesses the interface in a client domain solution. This is a laborious and error-prone job which increases exponentially when more systems are being updated: if two components, \( A_1 \) and \( B_1 \), are updated to \( A_2 \) and \( B_2 \), the application using these would potentially need four versions – one for the old interfaces using \( A_1 \) and \( B_1 \), and three for any combination of the component versions \((A_1 \text{ and } B_2), (A_2 \text{ and } B_1), \text{ and } (A_2 \text{ and } B_2)\).

### 3.3.3. Middleware domain

Middleware domain is the glue between the client application or components and server components in distributed systems. Shown as (c) in Figure 4, middleware acts as a mediator between the client- and the server side and thus all requests go through it in distributed systems. There may or may not be any middleware in non-distributed applications – a direct method call does not need any middleware. Well-known middleware services include CORBA [CORBA, 2002] [CORBA Components, 2002], RMI [Java 2 SE 1.4.2 Documentation, 2003] and Web Services [Wikipedia, 2008b].

In addition to basic middleware services, there exists a middleware mediator concept called enterprise service bus [Chappell, 2004], ESB, which is designed to connect heterogeneous services together. The greatest benefits of an ESB include that it is back-end agnostic – basically any server component can be integrated using an ESB. An enterprise service bus can support multiple versions of multiple components – there can be several ESB adapters that provide a different version of the interface and still connect to the same service instance.

### 3.3.4. Server domain

The easy and often used solution to the independent evolution problem is to use the binary compatibility rules of the target platform and it can be most efficiently done on the server domain (option (d) in Figure 4). Unfortunately, this typically leads to unmodifiable, immutable interfaces – at least there is no way to modify a method signature in an interface once it is published. The only way to change a method is to add another method with a different name or to create another interface that has the new method. Over time, there will be several partially overlapping legacy methods in interfaces that need to be supported just for backward-compatibility. This can be a big task and certainly degrades the quality of the code base, as there is a need to keep all the old methods up to date whenever the implementation is changed.
The server domain is the approach selected for this thesis but the approach is not using the binary compatibility aspects of a platform. Our solution is presented in Chapter 4. It provides a way to have freely evolutionary server components with externally multi-versional interfaces to the client components. The server domain comprises of the application server, the framework that runs the server components and provides the runtime environment to these components including the dynamic update capability, the container for multiple interface versions and the infrastructure for running them in parallel.

The benefits of solving the versioning problem at the server domain include the ability to run older versions of the clients as long as necessary while having potentially better-behaving applications due to the fact that they need to adhere to the framework and the services, which the application server forces on them. The disadvantages in turn include the fact that the server components must adhere to the provided framework and services – one cannot use a server domain solution to support applications not designed for the framework without modifications to the applications.

3.4. Goals

The goals for a system capable of running multiple versions of client applications for a server component are discussed in this chapter. Ideally, all goals should be fulfilled. In practice, however, for some environments it might be sufficient to partially meet the goals in order to get most of the benefits.

We have identified 11 goals and categorized them into three groups of requirements. The requirements directly related to dynamic component updates are described in sections 3.4.1 through 3.4.4, and sections 3.4.5 through 3.4.8 discuss the development- and run-time requirements. Finally, the non-functional requirements are detailed in sections 3.4.9, 3.4.10 and 3.4.11.

3.4.1. Dynamic update of the component implementation

An implementation update is considerably easier than an update of the whole component, in which the interface is updated as well, as the interface stays the same in an implementation update. Only the implementation part is changed, which does not affect the component interface.

The implementation of any component must be dynamically updatable without disturbing the system. This means that the system must serve clients even when the implementation is updated, i.e. at some point of time a client gets its request served by
the older version of the implementation and at the next invocation the client gets served by the new version of implementation.

Between these two points of time there must be no disruptions of service, the client must always receive service from either the old implementation version or the new implementation version.

3.4.2. **Dynamic update of the whole component**

A dynamic update of the whole component, an *upgrade*, involves modification of the interface and its implementation on the server side. This operation is complex as the clients depend on the very same interface that is now dynamically updated.

The goals for this operation are very much like the goals in the *dynamic update of implementation* but there are additional requirements. The dynamic update of the whole component *must not affect the clients still utilizing the old interface*. The server must provide service to client components using either the old interface or the updated interface.

The update of any component must be done without disturbing the system. The system must serve the clients using the old version of the interface all the time and start serving the clients using the new version immediately after the update is successfully completed.

3.4.3. **No modifications needed to the client components or systems**

The client components must be isolated from the changes to the server component and there must be no mandatory change in the client component due to the server component update. Furthermore, the system in which the client is running must require no changes when the server component is updated.

3.4.4. **State transfer support**

The system must support transferring the state from the old implementation to the new one during the update. The state transfer must be supported even when the whole component is updated so that the interface of the component changes.

3.4.5. **Multiple versions of interfaces concurrently used by the clients**

A server component must provide services to clients regardless of the versions of the interfaces, as long as such versions are installed in the system. The operation must be
concurrent, so that multiple client requests initiated by separate clients through different versions of the server component's interface can be run in the server component.

3.4.6. **Single running implementation serving several interface versions**

The system must allow a single implementation to serve requests from different versions of the interfaces of the component. This means that although the implementation is not implementing an older version of the component, the system must still allow the implementation to serve requests through the old interface.

3.4.7. **No constraints on modifying the interface**

There must be no constraints set by the system on how the old interface needs to be modified in order to provide a new interface and associated implementation. The system must not force to use version numbers in method calls or somewhere in the name of the interface. This means that it is not allowed to force the new interface to have a different fully qualified name from the old interface or to force a modified method to have a different name or signature (i.e. different parameters) from the original name or signature.

3.4.8. **No constraints on data types**

There must be no constraints on data types allowed in interfaces. All of the built-in types as well as custom types must be allowed. Even callback types must be allowed.

3.4.9. **System should not make development more complicated**

The development process for dynamically updatable components should not be *significantly* harder than developing components without the update capability. Some minor additional hurdles are allowed as the system as a whole makes the development easier by decoupling systems and components from each other. Linear growth of development work when number of server components increase is allowed but the number of client components must not affect to the amount of work.

3.4.10. **Performance must not degrade**

The performance of the server component in the system capable of running multiple concurrent interface versions must not be significantly lower than the performance of the server component in a traditional single-interface-version system. The client performance is not allowed to decrease either.
3.4.11. *Programming language and operating system independent*

The solution must be independent of operating system or programming language or environment.
4. MVCI framework

MVCI (Multi-Version Component Infrastructure) is a solution that provides an externally multi-versional component system. MVCI makes the component system seem multi-versional to the external systems and yet it only runs the latest version no matter what version the external system depends on. The external systems do not need to adapt to or even know that a different version of the component is active in the system than the one they depend on.

MVCI builds on the principle of strictly separating the component interface from its implementation. MVCI also introduces a concept of translator, which is used to translate component invocations from a version to another.

4.1. MVCI terminology

As all complex systems, there is a need for domain-specific terminology in order to successfully explain the MVCI system. The terminology is explained in details in Table 2.

<table>
<thead>
<tr>
<th>terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>application server</td>
<td>Server infrastructure running a set of components. Clients may either run inside the application server or be external to it.</td>
</tr>
<tr>
<td>component interface</td>
<td>A Component interface is an agreement between a client component and a server component. The formal component interface definition depends on the language and the platform used and it typically consists of header files (C and C++) or classes and interfaces (Java).</td>
</tr>
<tr>
<td>component implementation</td>
<td>The Component implementation is the part of the component that provides the implementation, the functionality of the server component specified by the component interface.</td>
</tr>
<tr>
<td>interface adapter</td>
<td>An interface adapter enables different versions of a component interface to use the same name space and clashing names within the name space. It handles the passing of the request from the name space of the old version of the interface to the name space of the new version of the interface to the interface translator. Interface adapter code can be automatically generated at development or deployment time.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>interface translator</td>
<td>An interface translator provides the full service described in the old version of the component interface, typically using newer versions of the same interface, or potentially totally different components and/or interfaces. Interface translators consist of both automatically generated and hand-written code and rely on the interface adapters.</td>
</tr>
<tr>
<td>component delegate</td>
<td>A Component delegate provides an indirection layer between the component interface and either the component implementation or an interface adapter. Component delegate makes it possible to dynamically switch the component implementation or interface adapter in use to another version of implementation or adapter.</td>
</tr>
<tr>
<td>server component</td>
<td>A component is in a role of a server component when its interface has been invoked by a client component.</td>
</tr>
<tr>
<td>client component</td>
<td>A component is in a role of a client component when it initiates the invocation to a server component.</td>
</tr>
<tr>
<td>interface registry</td>
<td>The interface registry is the directory of all existing versions of component interfaces of a component. The Interface registry keeps up the references to all interface adapters and component implementations for all versions of all components within one or more application servers.</td>
</tr>
<tr>
<td>component factory</td>
<td>The component factory is the application server's lookup and instantiation mechanism for components and versions of component interfaces. It uses the interface registry to perform its work.</td>
</tr>
<tr>
<td>effective version of component interface</td>
<td>The component interface backed by an implementation. In MVCI, there is always at most one effective version of component interface per component; other versions of interfaces are only used for supporting client components using old versions of the interface.</td>
</tr>
</tbody>
</table>

### 4.2. MVCI Components

A component is the basic building block for applications in MVCI. The applications are built by creating components and linking them together via their interfaces.
Components in MVCI consist of one or more version of one or more component interface, the component implementation, the interface translation layer and the packaging metadata. A component can be uniquely identified in the system by its name.

![Figure 5. Server Component and Packaging Metadata.](image)

Figure 5 shows a logical structure of a server component in MVCI. There are different versions of component interfaces \((A_1, A_2, B_1, B_2 \text{ and } B_3)\) connected to a single version of component implementation through an interface translation layer. A client component can use any version of any interface to access the services provided by the server component. The packaging metadata in Figure 5 is used by the MVCI framework to enable multiple versions of interfaces for a single component. It is used for the runtime configuration of the components, interfaces and the translation layer in MVCI.

4.2.1. Component interface

The component interface is a contract between a client- and a server component. The server component provides the services specified by one or more interfaces: the component interfaces must be implemented by the server component implementation. The only means for the client components or applications to access the services of the server component is via the server component interfaces. In Figure 5, the component interfaces are shown on the top (marked as \(A_1, A_2, B_1, B_2 \text{ and } B_3\)). In this case, there are two versions of interface type \(A – A_1\) and \(A_2 – \) and three versions of interface type \(B – B_1, B_2 \text{ and } B_3\).

A component interface consists of one or more interface definitions (for example Java interfaces or C++ pure virtual functions) that are implemented by the component implementation, and the interface-specific data type definitions (typically classes or structs) that are used to encapsulate the data passed between the client and the server via the component interface. There may be some simple logic in the component interface (such as helper functions to convert between data types) but the interface should never
contain application logic. The reason is that if the interface contains part of the
application logic, the maintenance of the application becomes very hard, as the
application logic cannot be updated independently of the interface. The application
logic should always reside in the component implementation.

As a contract between the client and the server, the component interface should remain
very stable – even immutable. Every modification of the component interface causes an
update not only to the server component but also to the client components. As the
update results in changes in the contract and the conditions, the interface should be as
stable as possible once it is deployed.

MVCI provides some flexibility to the immutable interface aspect by introducing
multiple versions of component interfaces. In MVCI, each version of the interface
should be immutable but changes are even encouraged between the versions if they
improve the application architecture. The multiple versions of a single interface make
also the contract situation between the clients and the server more interesting. The
server component is controlling the set of the versions of the interfaces available for the
clients. Thus, any given client must rely on one of the interface versions offered by the
server component. We can formulate the contract for the server component:

The server component must provide service for all interface versions it
defines.

And for the client component:

The client component must use one or more versions of the interfaces
provided by the server component to access the services on the server
component.

4.2.2. Evolution of the component interface

There exists no compatibility requirements for the different versions of the same
interfaces in MVCI. For example, in Figure 5, interface $A_1$ may be a subset of interface
$A_2$ (meaning that $A_2$ has all elements in $A_1$ supported, and potentially some more new
elements not in $A_1$, so that $A_2$ is fully backward compatible with $A_1$) but, on the other
hand, $B_1$ and $B_2$ may be totally unrelated so that there are no common elements at all.
Any of the claims in Figure 6 may be true in MVCI.
Claim (a) in Figure 6 is true if and only if the new and the old versions of the interface are identical. Claim (b) is true if and only if the new interface version contains everything in the old interface version and, in addition, something extra (such as a new function) while in claim (c), the situation is reversed and the new interface version is missing something that exists in the old version but brings no new elements. Claim (d) is only valid when the old and new interface versions have nothing in common and (e) is valid when there is something in common in the component interface versions but neither version is a subset of another.

Providing a framework that supports only cases (a) and (b) in Figure 6 would be trivial as all of the information provided by the old version of the interface version is also available in the new interface version and in exactly the same format, so it would just be the matter of forwarding the client's requests to the new interface (and component) version. The rest of the cases in Figure 6 are far more interesting as they certainly are not trivial. It is obvious that in cases (c), (d) and (e) the interface $A_1$ is not fully backward compatible with interface $A_2$ and thus cannot provide all the information needed by $A_1$. The missing information is addressed by the interface translation layer.

4.2.3. Interface translation layer

The interface translation layer provides automatic translation of interfaces so that it is sufficient to provide an implementation to a single version of an interface. In Figure 5, the interface translation layer provides the translation from interface $A_1$ to interface $A_2$, from interface $B_1$ to interface $B_3$ and from interface $B_3$ to interface $B_1$. This means that the component implementation only needs to support interfaces $A_2$ and $B_3$ and there is no need to make things more complicated by backing the legacy interface versions with implementation. Instead, the translation to the latest version is handled by the interface translation layer in isolation from the implementation.
In Figure 6, the system cannot generate the missing pieces of information for all possible invocations coming through the $A_1$ interface to the $A_2$ implementation in (c), (d) and (e) cases. Instead, either the interface translation layer is used to get the information from the other interfaces of the same or another component (for example interface $B_3$ may provide the missing pieces), or the translation layer can generate the missing information by computing the result or by sending a response that this information is not available (e.g. through raising an exception or by returning an error value).

Different versions of an interface can also have different structures, so that a version of an interface needs a single request to provide the service while another version needs two or more requests. The situation may also span multiple components and their versions. The problem can be addressed by splitting the requests to more requests or by combining the requests into fewer requests. Figure 7 depicts splitting (a) and combining (b) requests between interfaces and their versions.

![Figure 7. Splitting (a) and combining (b) requests.](image)

A request split means that during the system evolution, a function in an interface is decided to be split in two or more functions in one or more component interfaces. This interface split is reflected in the system so that the new version of the original interface no longer supports the same function as the old version. The disconnect between
interface versions is in this case addressed by using one or more functions of the new version of the interface, by using another interfaces of the component, by using the interfaces of totally different components, or by a combination of any of the previously mentioned solutions. The request split can be achieved in MVCI by using the translation layer to mediate the requests coming through older, not-yet-split functions to the relevant functions in the applicable interfaces of the correct components. Figure 7 (a) shows a situation where function 1 in the $A_1$ interface is split so that the system needs to invoke two functions in $A_2$ and one function in $B_3$ in the following order: $A_2$ function 4, $A_2$ function 3 and $B_3$ function 5. The request splitting is not necessary sequential as in the previous example – the split can be done based on the system state or the parameters of the functions as well – it can be a criteria-based split. The example in Figure 7 can also be interpreted so that with a certain input or system state the request to function 1 in the $A_1$ interface is forwarded to $A_2$ function 4, with some other input or system state to $A_2$ function 3, and with yet another input or state to $B_3$ function 5. There can also be a mix and match of the sequential and the criteria-based forwarding.

A request combination in turn means joining functionality of two or more functions of one or more interfaces to fewer functions in one or more interfaces. In Figure 7 (b), three functions in two different interfaces are combined into a single function of a single interface. The requests arriving to function 6 and to function 7 of interface $A_1$, and to function 8 of interface $B_1$ are combined to a single request to function 9 of the $A_2$ interface. The translation layer can wait for all the relevant requests ($A_1$ function 7, $B_1$ function 8 and $A_1$ function 6 in Figure 7) to arrive before invoking the target function of the target interface ($A_2$ function 9 in Figure 7). Similarly as with the splitting of requests, the combination of requests can be sequential or criteria-based, or a bit of both.

### 4.2.4. Component implementation

A component implementation contains the application logic for a single version of all interfaces that are supported by that specific component. The component implementation contains the logic for the latest version of the component interface only. The old versions of the interfaces are supported by the interface translation layer. The component implementation uses only the interfaces of other components to access the services provided by them. This way the component implementation automatically takes advantage of the interface translation layer of these other components when necessary.
In Figure 5, the component implementation provides the application logic for interface $A_2$ and $B_3$ and the translation layer supports the $A_1$, $B_1$ and $B_2$ interfaces. This means that the component as a whole (the component interfaces and their versions, the component implementation and the translation layer) serves the clients requesting service for any of these interfaces and their versions.

### 4.2.5. Packaging metadata

The packaging metadata contains the component metadata. The metadata consists of the component name, the interface names, the interface version number, the location of the executable code for the interfaces, the implementation version number and the location of the executable code for the implementation. Optionally, the metadata contains the details of the translators providing the translation from one interface version to another interface version.

The MVCI framework uses the packaging metadata to identify the component, its implementation and its interfaces. The adapter and the translator information of the metadata is used to set up the translation layer when a component is upgraded.

### 4.2.6. Using a component

In order to use a component, a client needs to locate a reference to the component using the component factory, the application server's component lookup service provided by MVCI. The client specifies the tuple \{component name, interface name, interface version\} to the lookup service in order to get a reference to the required interface of the component. MVCI instantiates the component and sets up all the required adaptation layers automatically for the component. After that, the client can use the services provided by the component.

### 4.3. Interface compatibility problem and solutions

In a complex distributed system it is common that a part of the system is updated and the rest of the system should work with the updated part. This means that the old interfaces of the components being updated are still used by the rest of the system during and after the update. We call this the *interface compatibility problem*. In this chapter we present three solutions to the interface compatibility problem. MVCI allows the utilization of any of the solutions described below.

#### 4.3.1. Traditional solution

The traditional solution to the interface compatibility problem is to keep the interfaces unchanged or at least backward compatible. The new functionality can be hidden
behind a new interface that the updated component implements in addition to the old one. We can write this as

\[ A_1 \subseteq A_2 \]

which means that the new interface \( A_2 \) is always equal to or a superset of the old interface \( A_1 \). This corresponds to the cases (a) and (b) in Figure 6 in chapter 4.2.2 and is to be interpreted so that \( A_2 \) is backward compatible with \( A_1 \), under the platform binary compatibility rules. The traditional solution provides limited support for request splitting and combination through allowing the application developers to invoke other components and functions in the component implementation part. The approach is laborious and tends to make the component interface and implementation harder to maintain.

The strictly controlled evolution of interfaces, due to the requirement for interface compatibility in the traditional solution, may lead to very complex component implementations that need to support truckloads of legacy interfaces. The approach severely limits the ability to re-architect a bad design decision.

### 4.3.2. Simple interface translation

A simple solution to the interface compatibility problem is to design a new interface independent of the old one and implement the old interface using the new one. In this way, the old interface uses the same implementation as the new one – albeit through the new version of the interface – and the redundant implementation is removed.

There needs to be a mechanism to translate the invocations of the old interface to invocations of the effective version of the interface (see Table 2 for terminology used). The improvement over using two separate implementations for the interfaces is that the actual implementation is in a single place. The rest of the code is just translator code. The simple interface translation fully supports splitting- and combining requests – the operations should be implemented in the translator code. The approach helps keeping the component interface and implementation clean.

There is a slight performance penalty involved in the translation process, but the major problem with this approach can be seen in Figure 8. The translators are interface-specific which means that a new translator must be written to all legacy interfaces whenever an interface is updated. In Figure 8 there are three legacy interfaces (a) that provide translation to the effective version of the interface. An upgraded interface (Interface v5) is introduced (b) and as the old translators can only use the version 4 of
the interface, they need to be rewritten to use the version 5 of the interface. The number of translator implementations needed grows exponentially as new interface versions are added. This also increases the size of the component packages, as every package needs to contain a translator for every single previous interface version.

![Diagram of component upgrade impact on translators](image)

**Figure 8. Component upgrade impact on translators. The interfaces and translators (a) before the upgrade and (b) after the upgrade.**

### 4.3.3. Transitive interface translation

The simple interface translation problems can be avoided by introducing a *transitive interface translation* mechanism. Figure 9 shows the concept in detail. A client connects to an older version of the interface (interface v1 in Figure 9) and sends a request to that interface of the component. The request is routed to a *component delegate* that forwards the request through the interface adapter to the interface translator (a) as the interface v1 is not the *effective version of the interface* and there is a newer version of the interface which is supported by the latest version of component implementation. The translator translates the request from Interface v1 to Interface v2 and forwards it to Interface v2 (b), which in turn forwards the request through the delegate, the adapter
and the translator (c), and all the way to the effective version of the interface (d) in Figure 9.

The Interface v3 is the effective interface of the component in Figure 9 and is thus backed by the component implementation. The request coming to the Interface v3 is forwarded through the component delegate to the actual component implementation (e). Return values are passed through the system in reverse order, in Figure 9 from Implementation v3 through the delegates, the interfaces and the translators all the way to the client that invoked the Interface v1. The interface translators perform the translation to the return values as well in the process.

The transitive interface translation chains up the different versions of the interfaces so that the old interfaces and translators can work as before when a new version of the component is upgraded to the system. One new node is added at the end of the chain. The upgrade package naturally needs to have the translator from the previous version to the current, effective version of the interface included. The transitive interface translation solutions supports both request splitting and combination in the translator, exactly as the simple interface translation solution.

When compared to the simple interface translation in Figure 8, the transitive translation in Figure 9 is a less labor-intensive approach than the simple translation. There is much more translation-specific implementation needed in the simple translation approach than in the transitive translation strategy, if the interface is upgraded more than once and the old interface versions still need to be supported.

It is possible to combine the transitive interface translation with the simple interface translation into a hybrid model, so that there is a direct jump from a certain translator to
a later interface in the chain. For example, if Interface v1 is used by many clients and there is a long chain of translations to the effective version of the interface, it is worth providing a direct translation from Interface v1 to the effective version of the interface as shown in Figure 10 (b). The interface translation may take some time especially if the chain of translators is very long but this is addressable by a strategically placed simple translator. In MVCI, the decision of the trade-off between the application performance and the developer productivity is left to the owner of the server component.

4.3.4. Evaluation of solutions

The traditional solution to the interface compatibility problem is very simple, requires no special support from the infrastructure and handles very well all of the special cases – such as callbacks, data types, etc. The challenge is that over time it tends to make the component interfaces complicated and hard to understand, as one is not allowed to change the existing definitions in the interfaces in a way that would break the backward compatibility.

Figure 10. Changing from transitive interface translation (a) to simple translation (b) may reduce the execution time spend in the translation process.

The simple interface translation and the transitive interface translation tackle the problematic areas unsolved in the traditional solution. In MVCI they can be both used when appropriate. If the transitive interface translation is used heavily, there is a chance that the execution time spent in the translation process increases too much. In these cases it is possible to introduce a simple interface translation to the specific old versions of the interface. In Figure 10, the transitive translation overhead from Client v1 to Server v3 in (a) can be reduced by introducing a simple interface translation between Interface v1 and Interface v3 (b), which saves one translation step.
Challenges in the translator solutions lie in certain special cases, where special attention is required to ensure the system performance with the interface translations, or to support callbacks (function pointers, pointers to remote objects), interface inheritance or custom data types defined in the interfaces (interface-private or shared). These special cases will easily make the framework quite complex. We will only take a cursory look at the special cases in the Reference implementation of MVCI-part in chapter 5, and discussions of other special cases are out-of-scope of this thesis.

4.4. Component versions in MVCI

The component versions are handled in a special way in MVCI due to the different approaches to updates and to upgrades. The component interfaces and the component implementation have separate version numbers. Updates and upgrades change different parts of the version numbers.

An update occurs when the component implementation is changed to another version and the interfaces are kept intact. An upgrade in turn involves modification of at least one interface so that at least one interface version is changed. An upgrade typically contains modifications to one or more component interfaces and to one or more component implementations. It changes the versions of the implementation and of one or more interfaces of the component. According to the terminology proposed by Cook and Dage [1999], an update introduces a new revision, while an upgrade introduces a new baseline version, a variant or a fusion. A revision is just a minor modification to the component, where the component interface stays backward compatible. A baseline version is a version of the component with an interface that is not backward compatible. Request splitting can be supported by a variant, and a fusion supports combining requests.

4.4.1. Version notation for MVCI

A version notation identifies the versions of the interfaces and the implementation. It is represented as \{i\}{i\_version : x}, where i is the name of the interface, i\_version is the version of the interface and x is the version of the implementation. This makes it easy to distinguish a dynamic upgrade where the interface version of at least one component changes from a dynamic update where the interface stays the same and only the component implementation changes. The version notation essentially describes the interfaces that can be used to connect the component, and the version of the implementation. The notation can be extended to \{i\}{i_1, i_2, ..., i_n : x} when a component has more than one interface versions and to \{i, j, ...\}{i_1, i_2, ..., i_n; j_1, j_2, ..., j_n; ... : x} when the component has more than one version of more than one interfaces.
As an example, Figure 9 in chapter 4.3.3 has the version

{Interface} \{v1, v2, v3 : v3\}

If a component, for example, has three interfaces named $A$, $B$ and $C$, each of them has three versions ($A_i$, $A_2$, $A_3$; $B_2$, $B_3$, $B_4$; and $C_3$, $C_4$, $C_5$; respectively) installed and the implementation version is 3.23, this would be

{\{A, B, C\} \{A_1, A_2, A_3; B_2, B_3, B_4; C_3, C_4, C_5 \}} : 3.23

in the MVCI version notation.

### 4.4.2. Updating the implementation

The dynamic update case where only the implementation part is updated is very straightforward. An update of the component implementation from the version {Interface} \{v1 : v1.0\} to the version {Interface} \{v1 : v1.1\} is depicted in Figure 11.

The process of updating is:

1. The running implementation part is stopped from receiving any new service requests (case (b) in Figure 11) by queuing the requests in the component delegate
2. The outstanding service requests on the component implementation are allowed to finish
3. The state of the component is stored in a persistent storage
4. The component implementation is stopped and removed from the memory
5. The new implementation version is started
6. The component state is restored from the persistent storage
7. The service requests (including the ones pending in the queue of the component delegate) are routed to the new implementation version (case (c) in Figure 11)
In the update process, the interface part stays the same up to the component delegate, which is retargeted to the new component implementation. The system can also start the new version in parallel to the shutting down of the old version if the component does not need to store its state and the different versions do not compete over same resources. This allows rapid transition to the new implementation as there is no need to wait for the old implementation to shut down.

4.4.3. **Upgrading the interface**

The process of upgrading the whole component including its interface is a more complex one. The old version of the interface must be allowed to continue serving requests from the older clients. The component delegate provides the required mediation behind the old interface to achieve this. Translators are then used to implement the logic to translate the requests from an older version to a newer version of the component. Figure 12 shows what happens in an MVCI system during the upgrade from \{Interface\} v1 : v1.0 to \{Interface\} v1, v2 : v2.0.
The update process is following (Figure 12):

1. The running implementation part is stopped from receiving any new service requests (a) by queuing the requests in the component delegate
2. The outstanding service requests on the component are allowed to finish (a)
3. The state of the component is stored in a persistent storage
4. The component implementation is stopped and removed from the memory
5. The translator for the old interface is initialized and started in the place of the old implementation (b)
6. The new versions of the interface and the implementation are started (b)
7. The new implementation restores the state of the old implementation from the persistent storage
8. The translator from the older version is targeted to the new interface (b)
9. The service requests are enabled on the new interface
10. The service requests are re-enabled on the old interface and the requests queued in the component delegate are routed to the translator

The upgrade process is much heavier than a simple implementation update as there is the need to set up potentially many interface translations from the old versions to the new version of the interface. In Figure 12, a component with the version \{Interface\} {v1 : v1.0} is upgraded to \{Interface\} {v1, v2 : v2.0}, which means that the component has the interface versions v1 and v2 available to the clients while the implementation version is v2.0.
An upgrade, which changes the whole interface structure of the component, can be handled in the same way as an upgrade, which only changes the interface version. The system supports translators translating from an interface to a totally different interface of the component by the means of having the translators acting as client components to the target interfaces. In this case the versions would change from \{A^1, A^2, ..., A^n\} \{A^1_\text{versions}; A^2_\text{versions}; ...; A^n_\text{versions} : \text{old\_version}\} to \{A^1, A^2, ..., A^n, B^1, B^2, ..., B^m\} \{A^1_\text{versions}; A^2_\text{versions}; ...; A^n_\text{versions}, B^1_\text{versions}; B^2_\text{versions}; ...; B^m_\text{versions} : \text{new\_version}\} where

\[ \{A^1, A^2, ..., A^n\} \cap \{B^1, B^2, ..., B^m\} = \emptyset \]

This means that the new component version directly supports none of the interfaces of the old version of the component. The new component would still support the old A^1, A^2, ..., A^n interfaces but only via translation to the new B^1, B^2, ..., B^m interfaces.

4.4.4. MVCI versions – the client view

The clients do not see the different versions within MVCI; they merely use the version of the interface they need. A client does not need to know anything about the MVCI version notation or the MVCI version numbering other than what is the name of the component, the name of the interface and the version of the interface required. Everything else is hidden from the client.

A client needs to place a request to the application server's component factory with a version number of the server component interface the client is accessing in order to get a reference to that component. The client actually gets a reference to the component delegate with the requested interface version and from there on the request is routed to the implementation or to the translation layer.
5. Reference implementation of MVCI

This chapter describes our reference implementation of MVCI in the Java programming language. There is nothing preventing from choosing another platform – our selection of the Java platform is only based on the fact that we are very familiar with the language and the platform.

The MVCI reference implementation is far from a perfect implementation of the MVCI framework described in chapter 4. We will take a look at the supported features and the feature omissions in section 5.1, and the environment on which the MVCI reference implementation runs.

The components must be packaged in a special JAR file [JAR File Specification, 1999] in the MVCI reference implementation. The structure of the JAR file and its relation to the interface versions, the component implementation versions, the adapters and the translators are discussed in section 5.2.

The MVCI reference implementation depends heavily on dynamic library loading and unloading. This is handled by class loaders in Java. The MVCI reference implementation uses a special class loader hierarchy to achieve the goal of having externally multiple versions of the component interfaces available to the clients. We elaborate on the class loaders, and discuss how they are used and how they are tied with the packaging format in section 5.3.

Full source code for the MVCI reference implementation is available in Appendix G. The license for the MVCI source code can be viewed in Appendix D and E. Appendix F contains brief instructions for unpacking the sources as well as short usage instructions of the MVCI reference implementation.

5.1. Description of the reference implementation

As our MVCI implementation is a proof-of-concept with the sole goal of supporting the development of the MVCI architecture introduced in chapter 4, there are certain omissions in the implementation as well as features that are differently or not fully implemented as described in the general MVCI framework section of this thesis.

5.1.1. Features and omissions

The MVCI reference implementation is capable of running multiple components in parallel. There may be zero or more client applications, components that are only in the client role – these are implemented mainly for system testing purposes. The amount of
server components is not limited by the implementation and they can also act as client components to other server components. The implementation supports dynamic component installations, updates and upgrades but the uninstall operation is not supported. Furthermore, the installation state of the components is not preserved over the system restarts so the components need to be reinstalled every time the system is started. The installation and update operations are done by using a (very pragmatic) GUI that is built-in to the system and is started automatically with the system.

The reference implementation fully supports running multiple versions of interfaces in parallel and a request to any interface version is forwarded to the single component implementation. The number of different interfaces of a component is limited to one as it is enough for this proof-of-concept.

The only supported solutions to the interface compatibility problem (see chapter 4.3) are the transitive translation and the traditional solution. The simple translation solution is not supported because a new component version can only have a single translator that translates from older versions in our implementation. Simple translation solution would require multiple translators per component version. Related to this, support for the automatic generation of the adapter and the translator is not implemented either.

The MVCI reference implementation does not support state transfer from an old version of a component to a new version that supersedes the old one. The state transfer between component versions is not in the scope of this thesis. The reference implementation of MVCI runs all components locally in a single Java VM. Thus, distributed computing is not enabled in the reference implementation – but it would be quite easy to extend the MVCI reference implementation to support the distributed computing model.

The reference implementation does not support deadlock detection or prevention. A deadlock could happen during the upgrade operation with three components, A, B and C where A invokes B which in turn invokes C. At this point, component B is upgraded which means that the system is waiting for all of the ongoing operations in the implementation of B to finish before the new version of B can be started. If, at this point, C invokes B, we have a deadlock situation where the invocation is waiting for another invocation deeper in the call stack to finish, which in turn is impossible before C \rightarrow B invocation is finished.

5.1.2. Runtime environment of MVCI reference implementation

The reference implementation of MVCI relies on a standard Java platform as defined in the Java 2 SE 1.4.2 Documentation [2003]. Any Java SE 1.3 – 6.0 release [Java SE,
should be able to run the MVCI reference implementation and there is no limitation on the operating system either (we've run MVCI successfully on Microsoft Windows, FreeBSD, Linux and Mac OS X).

The only external library needed in addition to standard Sun Java SE SDK [Java SE, 2008] is Ant [Apache Ant, 2008] and it is only needed for building the MVCI reference implementation from sources. As a convenience, there is an Ant target to run the MVCI reference implementation as well.

5.2. Packaging and metadata information

The components are packaged in a single JAR files in MVCI with metadata in the manifest [Java 2 SE 1.4.2 Documentation, 2003] [JAR File Specification, 1999]. This allows MVCI to have a simple packaging format that uses and extends the well known JAR format. The structure of the JAR file is defined in this chapter. The MVCI reference implementation uses nested JAR files to contain the different parts of a component (the interface, the implementation, the adapter and the translator) and a JAR manifest to provide the metadata of the component.

5.2.1. MVCI manifest content

In MVCI, the component metadata is kept in the JAR manifest. There are a number of parameters required to provide the automatic version translation. Specifically, what the implementation needs to know about the component JAR file is:

1) The name of the component
2) The versions of the component and its interface
3) The older version of the interface which is being translated by the component
4) The names of the JAR files inside the component JAR file providing the class files for (a) the implementation, (b) the interface, (c) the translator and (d) the adapter
5) The fully qualified names of the entry-point classes for (a) the interface, (b) the implementation and (c) the translator.

The detailed metadata is illustrated in Table 3. The first nine (from Name to Adapter-Jars) parameters are mandatory for every single version of a component. The rest three (Translator-From-Interface-Version, Translator-Jars and Translator-Class) are only mandatory if the version of the component in question contains a translator from an earlier version of the component.
We are creatively misusing the JAR manifest individual section Name [JAR File Specification, 1999] in the MVCI reference implementation design. The set of MVCI manifest parameters must start with the individual section Name and the field must be set to value mvci.component. This actually contradicts with JAR File Specification [1999] but works with all Sun Java 2 SE implementations at least from 1.3 to 6.0.

The (abuse of the) individual section allows the MVCI reference implementation to handle the MVCI manifest parameters as an individual set of manifest entries. The MVCI implementation needs only to look for the individual section with the name mvci.component in order to nicely get all the parameters defined for the component – there is no need to scan through the whole manifest. There is a limitation, though: only one component can be defined in a single JAR file.

The component name in Table 3 uniquely identifies the component in question and it actually corresponds to the interface name in the MVCI reference implementation as there can only be a single interface for a component. The interface version refers to the interface which is included in the component JAR file and which is backed by the component implementation. If the component supports other interface versions, they are dynamically collected from the existing versions during the upgrade operation by using the Translator-From-Interface-Version -parameters in the manifests of the components. Appendix A contains an example of a manifest metadata.
<table>
<thead>
<tr>
<th><strong>Field Name</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>The attribute name for MVCI component. Value must always be <strong>mvci.component</strong>.</td>
</tr>
<tr>
<td>Component-Name:</td>
<td>The name of the component.</td>
</tr>
<tr>
<td>Interface-Version:</td>
<td>The version number of the component interface.</td>
</tr>
<tr>
<td>Interface-Jars:</td>
<td>A comma-separated list of names of the JAR files containing the component interface.</td>
</tr>
<tr>
<td>Interface-Class:</td>
<td>The fully qualified class name of the component interface that the component implementation supports.</td>
</tr>
<tr>
<td>Implementation-Version:</td>
<td>Version number of the component implementation.</td>
</tr>
<tr>
<td>Implementation-Jars:</td>
<td>Comma-separated list of names of the JAR files containing the component implementation.</td>
</tr>
<tr>
<td>Implementation-Class:</td>
<td>The fully qualified class name of the component implementation entry-point class that implements the interface defined in Interface-Class.</td>
</tr>
<tr>
<td>Adapter-Jars:</td>
<td>A comma-separated list of names of the JAR files containing the interface adapter.</td>
</tr>
<tr>
<td>Translator-From-Interface-Version:</td>
<td>The version number of the component interface the interface translator provides translation from.</td>
</tr>
<tr>
<td>Translator-Jars:</td>
<td>A comma-separated list of names of the JAR files containing the implementation of the interface translator.</td>
</tr>
<tr>
<td>Translator-Class:</td>
<td>The fully qualified name of the interface translator entry-point class that handles the incoming translation requests from old interface version through the interface adapter.</td>
</tr>
</tbody>
</table>
5.2.2. Component packaging

One must use JAR files inside the component JAR file as the packages for the component interface-, the component implementation-, the interface adapter- and the interface translator class files, i.e. the component implementation must be packaged into one or more JAR files so that they do not contain any interface-, adapter- or translator class files. These JAR files must be included in the component JAR file. The same goes with interface-, adapter- and translation classes. The restriction for the contents of the JAR files is included because of the way the Java class loaders work: if the implementation class is loaded by the interface class loader there is no way of unloading the implementation without unloading the interfaces and all clients using the interface in Java (we will discuss this further in chapter 5.3), which is exactly what we're trying to avoid with MVCI.

Table 4. Contents of an example component JAR file.

<table>
<thead>
<tr>
<th>JAR File Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>META-INF/MANIFEST.MF</td>
<td>The manifest file containing the metadata for the component.</td>
</tr>
<tr>
<td>c2c3translator.jar</td>
<td>The class files for the interface translator from the interface version 2 to the version 3.</td>
</tr>
<tr>
<td>c3adapter.jar</td>
<td>The class files for the interface adapter of the interface version 3.</td>
</tr>
<tr>
<td>c3impl.jar</td>
<td>The class files for the component implementation of the version 3 of the component interface.</td>
</tr>
<tr>
<td>c3inf.jar</td>
<td>The class files for the component interface version 3.</td>
</tr>
</tbody>
</table>

Table 4 shows the structure of a component JAR file from a sample component of the MVCI reference implementation. The component provides version \{component1\}{3 : 3.0} and contains a translator from version 2 of the interface. If the component is upgraded on a system, which includes the version 2 of the interface, the component version will become \{component1\}{2, 3 : 3.0}, or potentially \{component1\}{1, 2, 3 : 3.0} if the version 1 was installed to the system. In Table 4, the MANIFEST.MF file in the META-INF folder contains the metadata information for the component, c2c3translator.jar contains the translator from the interface version 2 to the version 3. The adapter is included in c3adapter.jar and the interface is in c3inf.jar. The implementation resides in c3impl.jar.
In Table 4, the different parts are packaged in separate JAR files inside the component JAR file. The MVCI reference implementation allows using several JAR files for the class files of each part – the interface, the adapter, the translator and the implementation (for example the implementation could consist of three different JAR files inside the component JAR). These files may generally not be shared across the different parts of the component. Contents of the JAR files of the sample component are available in Appendix B.

5.3. Java class loaders in MVCI

The MVCI reference implementation relies on dynamic loading and unloading of classes for the interfaces, the adapters, the translators and the implementation. The dynamic loading is essential to MVCI, without it there would not be any dynamic updates. To achieve dynamic loading in the MVCI reference implementation, we're using Java class loaders.

Java language [Joy et al., 2000] has a special means for allowing dynamic loading of class libraries using special Java objects: class loaders. Class loading functionality allows lazy loading, type-safe linkage, user-definable class loading policy and multiple namespaces [Liang and Bracha, 1998].

Lazy loading means that classes are loaded on demand, the classes are only loaded when needed and not before. This reduces memory usage and improves the system response time. Type-safe linkage ensures that the dynamic class loading does not violate the type safety of the Java language. The type checking is not done at runtime as it would deteriorate the runtime performance; instead it is done at the dynamic linkage phase. User-definable class loading policy gives the programmers complete control over class loading including the source of the classes and the ability to modify the loaded classes at runtime by adding, for example, security attributes to the classes. Multiple namespaces allow separation of components that are running simultaneously. Utilization of multiple namespaces makes it possible to disable the access from a component to the methods of another component in another namespace. [Liang and Bracha, 1998]

The ClassLoader Java class uses a delegation model to search for classes and resources. Each instance of ClassLoader has an associated parent class loader. When requested to find a class or a resource, the ClassLoader instance will delegate the search for the class or for the resource to its parent class loader before attempting to find the class or the resource itself. The virtual machine's built-in class loader called the bootstrap class
loader does not itself have a parent but may serve as the parent of a ClassLoader instance. [Java 2 SE 1.4.2 Documentation, 2003]

In Java, a class type is uniquely determined by the combination of the class name and the class loader instance that loaded the class [Liang and Bracha, 1998]. This means that classes loaded by different class loaders are not able to directly reference to each other, other than by their supertypes loaded by a parent class loader common to both of the class loaders, or via Java reflection [Java 2 SE 1.4.2 Documentation, 2003]. According to Liang and Bracha [1998], a class cannot be unloaded unless its class loader is garbage collected. In order to allow dynamic updates, we need to be able to unload classes and thus must use class loaders. Otherwise, over the time the system memory would become filled with old classes that are no longer used for anything.

5.3.1. Class loader relations in MVCI

The MVCI reference implementation is using Java class loaders to load and unload interfaces, implementations, adapters and translators. Several class loaders are needed per component in the MVCI reference implementation in order to isolate the component elements from each other in a way that makes updates and upgrades possible. The class loader hierarchy in the MVCI reference implementation is shown in Figure 13.

In the MVCI reference implementation we are using two types of relationships between class loaders. The basic class loader relation, the parent-child relation, allows the classes loaded by the child class loader to directly access the classes loaded by the parent class loaders. This allows the system class loader to load all of the Java system classes and lets the classes loaded by a custom class loader automatically use the system classes. The classes loaded by the custom class loader can be unloaded independently of the classes loaded by the parent class loader. The relation between Interface $B_x$ class loader and Implementation $B_x$ class loader in Figure 13 is a parent-child -relation where Interface $B_x$ is the parent class loader of Implementation $B_x$.

Unfortunately, the parent-child relation does not solve all of our problems in the MVCI reference implementation. We need a uses relation in order to provide the client component the access to the server component. A class loader can only have a single parent class loader and the hierarchy cannot be changed dynamically so the class loader of a client component that uses several server components cannot have the server components' class loaders as the parent class loaders. In order to access the server components' interfaces the client component's class loader needs to be able to access the class loader that loaded the interfaces.
To solve this component interface access problem, we have created a custom class loader that is capable of using other class loaders (a uses relation). This is achieved by having a dynamic list of friend class loaders in the custom class loader. If the class is not found by the parent class loaders or by the custom class loader itself, the list of friend class loaders is used to load the class. The MVCI reference implementation is dynamically adding and removing friend class loaders to and from the custom class loader's list in order to allow the access to the component interface for the client components and for the translators as well. The relation between Translator $A_{1,2}$ class loader and Interface $A_2$ class loader in Figure 13 is a uses relation.

![Figure 13: Class loader hierarchy in MVCI. There are two versions of interfaces of component A and a single client component (B) that uses the component A.](image)

### 5.3.2. Class loader architecture in MVCI

Figure 13 shows the class loader hierarchy in the MVCI reference implementation. We have depicted a situation where component $A$ has two concurrently running interface versions, Interface $A_1$ and Interface $A_2$. There is also a translator in work between the old and the new version of the interface. Translator $A_{1,2}$ provides – in concert with Adapter $A_1$, Interface $A_1$ and Interface $A_2$ – an automatic translation to the new interface version for clients still using Interface $A_1$.

The interface needs its own class loader which in turn is used by the implementation in a parent-child relation, and by the translators and the clients running in the same application server instance in a uses relation. This makes invocations between components running in the same application server very efficient as there is no need for any marshaling of the parameters and the method signatures, which would be needed if
invoking a remote component or a component residing within a different class loader space inside the same Java virtual machine without a proper class loader hierarchy. Each interface version has its own class loader instance which makes it possible to have several different versions of an interface running simultaneously in single Java virtual machine without any name clashes in the MVCI reference implementation. In Figure 13, Interface A₁, Interface A₂ and Interface B, represent the interface class loaders. Their parent class loader is the MVCI framework class loader, which loads the MVCI application server.

A component implementation needs a class loader as well in order to separate the component implementations from each other and to enable the dynamic update of the implementation. The implementation class loader is using the class loader of the interface it implements as the parent class loader and, thus, to load the classes of the interface. This makes it possible to unload the implementation without unloading the interface. It is necessary to be able to load and unload the versions of the implementation independently of their interface because it is the only way to isolate the client components from the impact of changing the implementation version of the server component. In Figure 13, Implementation A₂ and Implementation B, represent implementation class loaders.

The adapter class loader is used to separate the adapter from the interface namespace. The adapter class loader is the parent of the translator class loader, which in turn is using the translation destination interface class loader. As there may be clashing class names in the translation source and destination interfaces, the translator class loader cannot directly use both class loaders of the source- and the destination interfaces. The adapter handles the conversion from the source interface class loader namespace to the translator namespace while the destination interface namespace is directly accessed through a uses relation between the translator class loader and the destination interface class loader. Adapter A₁ and Translator A₂ are parent and child class loaders, respectively, in Figure 13 and thus the adapter classes are accessible from the translator, but not vice versa. This class loader setup would allow independent dynamic updates of the translators as well. The translators are at least partly hand-coded and there is a chance that an update is required but the MVCI reference implementation does not support dynamic translator updates. The adapters, in turn, are generated from the component interface so their update cycle is tied to the interface update cycle. Separate dynamic adapter updates are not needed.

There is no relation between the component interface class loader and the adapter class loader. Instead, the Java reflection [Java 2 SE 1.4.2 Documentation, 2003] is used to
dynamically transfer method invocations from a class loader's namespace to the other's namespace. An invocation handler [Java 2 SE 1.4.2 Documentation, 2003] – that is a component delegate in Java language – is installed for each interface in the MVCI Framework class loader namespace. The invocation handler enables the retargeting of the invocations to the interface adapter when a new version of the component interface is upgraded. The invocation handler uses the Java reflection [Java 2 SE 1.4.2 Documentation, 2003] to forward the invocations to the interface adapter and to its namespace. In Figure 13, this is drawn with the arched connector between the *Interface A* and the *Adapter A* class loaders.

### 5.3.3. Relation between the server- and the client component

In figure 13, component B uses the services of component A – component B is actually invoking methods of the 2nd version of component A’s interface. It is shown as an arrow from the Implementation B, class loader to the Interface A class loader. A *uses* relation connects the Interface A and the Implementation B, which means that Implementation B, uses the classes of Interface A. This link is set up at runtime when the client component needs to use the interface of another component in the same application server.

Note that system does not prevent a client component from using the older version of the server component's interface, so the arrow from *Implementation B* could go to *Interface A* rather than A. The translation layer would take care of translating the invocation and return values from *Interface A* to *Interface A* and vice versa.

We have designed the class loader hierarchy so that it isolates client components from server component implementations. The client components only have the access to the interface of the server component; they do not have any direct access to the classes or the methods of the server component implementation. This arrangement makes it possible to switch the implementation to another version without any effect on the client components.
5.4. Interface translation in action
The interface translation in the MVCI reference implementation requires a number of different parts to play together. The component interface is the first point of contact for a method invocation by a client. The invocation is passed to the interface adapter through the component delegate. The interface translator implements an interface of the interface adapter, gets the invocation from the adapter and can then perform the actual translation to another version of the interface by simply accessing the types and the methods of the new interface.

5.4.1. Component interface – component delegate – interface adapter
The MVCI reference implementation provides a component delegate – a Java reflection layer – between the interface and the interface adapter. The component delegate transforms component interface invocations to interface adapter invocations when a translation to another version of the interface is needed. The component delegate is designed to make use of automatically generated adapters and relies on certain conventions in transforming the invocations. An adapter consists of a renamed interface where the package name of the interface is prepended with the version number of the interface. This way the adapter class names do not clash with the translation target interface class names. The arrangement is necessary because the adapter and the translator utilize the namespace of the translation target interface.

Because the adapter interface is identical to the component interface, it is easy to identify the correct method to be invoked in the adapter as it has a similar signature as the invoked method has in the component interface. Translating the parameters, exceptions and return values is somewhat complicated and we will discuss about that in detail in section 5.4.2.

In the sample component in Appendix B, the version 2 of the ComponentOne interface is to be translated to the version 3 of the interface (ComponentOne\{2, 3 : 3.0\}). The adapter for the interface version 2 is identical to the interface but the package name is prefixed with '_v2'. The component delegate forwards the invocation from the

```java
public void invoke(long key,
        fi.uta.joonashaapsaari.compol.Payload data) throws
        fi.uta.joonashaapsaari.compol.PayloadException;
```

method of the

```java
fi.uta.joonashaapsaari.compol.ComponentOne
```
interface to the

```java
public void invoke(long key,
        _v2.fi.uta.joonashaapsaari.compol.Payload data) throws
        _v2.fi.uta.joonashaapsaari.compol.PayloadException;
```

method of the

```java
_v2.fi.uta.joonashaapsaari.compol.ComponentOne
```

interface of the interface adapter. This method is then implemented by the interface translator and it is the translator's responsibility to translate the method invocation to version 3 of the ComponentOne interface.

5.4.2. Handling parameters, exceptions and return values – interface adapter

When the component delegate forwards the request from a component interface to the interface adapter, it needs to utilize the Java reflection mechanism, as the interface and the adapter are in a different namespace. The different namespaces also mean that any parameter, return value or exception defined in the component interface needs to be dynamically copied to the interface adapter's namespaces. The MVCI reference implementation uses a (bit barbaric) brute-force method to achieve this.

The following procedure for copying parameters from a component interface to its interface adapter is used:

1. Every source type in the parameter list is gone through one-by-one.

2. If the source type can contain other types, each of the type is gone through one-by-one similarly as the parameter list.

3. If the source type is not loaded by the component interface class loader, it is copied verbatim as a destination type to the list of interface adapter parameters. The class of the source type is common to both of the namespaces, thus, there are no clashes in names.

4. If the source type is loaded by the component interface class loader, a type with the same name but with a package prefix representing the interface version is created in the interface adapter namespace. The fields are copied from the source type to the destination type in the similar way as the whole list of parameters is gone through.
The automatic copying method described above must be repeated for the return value and the exceptions coming form the target interface. Of course, the source and destination class loaders and namespaces are reversed as the types arriving from the newer interface need to be adapted to the older interface in this case.

This method is somewhat computationally laborious and time-consuming if the list of parameters is very large and contains a lot of types defined in the component interface. Lists, arrays and sets of elements are particularly computationally-intensive as every entry must be gone through in the list. The current implementation of MVCI only supports shallow copying of fields within a type and will not work for arrays, other collections of objects or types containing deep structures.

The problem with copying parameters is highlighted in strongly typed platforms such as Java. For example, in C language, the parameters could just be copied verbatim without any adaptation as the language is weakly typed and the parameters are handled merely as pointers to a memory location. A better method for the parameter copying for Java – be it a dynamic lazy one where the translation is done only when necessary or something totally different, perhaps related to the Java Virtual Machine implementation – is an excellent candidate for further study.

5.4.3. Translator

The translation itself is a quite simple process of adapting old interface requests to requests to the new interface version. In practice, the translator must extend the AbstractTranslatorBase -class (provided by the MVCI reference implementation) and implement the interface adapter's interface corresponding to the main interface of the old version of the component interface. The delegate then automatically invokes the translator and it is the translator's responsibility to invoke the new version of the interface. The reference to the new interface (and either the implementation or another adapter-translator structure) is set up to the target field of the AbstractTranslatorBase.

As the old interface version differs from the new interface version, it is generally not possible to automatically provide the translation. Certain parts could be automated but that is not in scope of this thesis (but is yet another candidate for further study) – the MVCI reference implementation does not support any automatic translation. The translator implementation must copy all parameters from the types defined in the adapter interface to the types defined in the new version of the component interface. After that, the translator must invoke the correct method(s) on the new interface.
version. Finally, the return values and exceptions need to be copied back to the adapter types.

5.5. Different types of reconfiguration operations

In MVCI, there are four basic types of component reconfiguration operations. These are

1. Installation of a component
2. Update of the implementation of a component
3. Upgrade of the whole component
4. Uninstallation of a component

A new component is added and configured in the installation operation. This involves adding the component binaries to the system and configuring the system so that the new component is usable for its clients. The interface adapter for the component is configured in the installation operation as well but it does not play any role until the component is upgraded.

An update operation changes only the implementation of the component being reconfigured. This operation is useful for example in a situation where there is a software error – a bug – in the component implementation. The interface does not need to be changed at all and thus clients can continue using the same interface after the reconfiguration. The old implementation will no longer receive invocations after the reconfiguration; the invocations are rerouted to the new implementation of the same interface.

On upgrade operation, the whole component is changed including its interface and implementation. The translator translating from a previous version of the interface to the current version is also added to the system. The old implementation will no longer receive invocations after the upgrade. The old interface version may receive invocations but they are rerouted to the adaptation and further to the new interface version. An interface adapter for the upgraded version of the interface is installed as well.

Uninstallation means completely removing the component from the system. The MVCI reference implementation does not support uninstallation.

The different operations needed in MVCI reference implementation are automatically detected based on the system state. The system state is read from the component registry that is keeping books on all components and their versions.
5.5.1. Component registry

The key element in the MVCI reference implementation during a reconfiguration operation is the component registry that is used to store the MVCI-specific metadata of the components. It also keeps up the references to the running components so that the client components can locate the server components by using the component factory.

The component registry contains references to the component delegates for all versions and to all different class loaders of interface versions including the interface class loaders, the adapter class loaders, the translator class loaders and the implementation class loaders. The component registry has information on the effective version of the component interface, on the installed versions of a component and on how to get a reference to the component delegate of any of the versions. In short, the component registry is the information storage for the reconfiguration operations of the system and for the component version reference lookup for the clients.

5.5.2. Installation

The MVCI reference implementation automatically detects that an installation is needed by searching for the component in the MVCI component registry. An installation operation is in question if the component name is not registered or no existing version under the component name is found in the registry.

Installation involves reading the component JAR file, unpacking the JAR file and putting the contents in places where the relevant interface-, adapter-, translator- and implementation class loaders can find them. In addition, the component delegate and the implementation classes need to be initialized and put to the component registry along with other metadata so that clients can find the reference to the component and start using the services provided by it.

5.5.3. Implementation update

An implementation update involves reading the component JAR file similarly as in the installation phase. The MVCI reference implementation detects that the operation is an update operation by comparing the interface- and the component versions in the component registry and in the component JAR file under reconfiguration. If the interface version of the component JAR file is equal to the interface version of the currently running component, and the implementation versions of the JAR file and the running component are not equal, the framework can conclude that an update operation is required.
On the update operation only the implementation JAR file inside the component JAR file is extracted and a new implementation class loader instance is initiated for it. The new implementation class is initialized and it is registered to the component repository along with the new implementation class loader under the existing interface object replacing the data referring to the old implementation. The component delegate is kept but the reference to the implementation object it contains is updated to point to the newly added component implementation. Thus, all new invocations to the component will end up in the new implementation object.

5.5.4. Component upgrade

A Component upgrade requires the MVCI reference implementation to configure a new version of the component interface. The need for an upgrade is determined by searching the interface versions from the component registry and by comparing those to the interface version in the JAR file manifest. The reconfiguration operation in question is an upgrade if

1. there is no existing interface for the component with the same interface version as the JAR file manifest has in the component registry, and
2. there is a translator in the JAR file manifest that has a source interface version that matches to the effective interface version in the component registry

The interface, the adapter, the implementation of the new component version and the translator for a previous version of the interface are unpacked from the component JAR file. The old implementation is stopped and the translator is wired to take its place along to the adapter, which was already installed with the previous version of the component. The new version is then installed after which the translator from the old version is targeted to the component delegate of the new version. The component registry is updated to reflect the new state of the component. After that the requests are allowed for the new and the old interfaces.

5.6. Performance of MVCI reference implementation

In this chapter we're going to discuss the performance of the MVCI reference implementation. We are going to focus on two aspects, namely the developer performance when developing on the framework and the application performance with the automatic interface translation in use.

5.6.1. Developer performance

Supporting the automatic interface translations in the MVCI reference implementation requires the developers to perform some extra work in addition to the regular
application component development. For simplicity, we are assuming that the developers would develop on a framework similar to the MVCI reference implementation, although without the support for multiple versions. The basic idea behind that is that we believe that most of the aspects in the MVCI framework can be incorporated into the mainstream application servers – a topic for further study.

Without the multiple version support, the developers would need to define the components – the interfaces and the implementations – and the packaging metadata. On the MVCI reference implementation, one will need adapters for all versions of the components and translators for the components that need to support multiple versions of interfaces. Additionally, some extra metadata would be required for all of the components. The generator for the interface adapters is missing but it should not be a huge task to develop one so we assume here that an adapter generator would be available if the MVCI framework would be taken into use.

In the end, what needs to be done by the developers is to add a small amount of metadata, which is quite trivial, and some translator code for the upgraded components. We estimate that the extra effort required by the MVCI reference implementation is relatively small compared to the advantages it will give in a complex distributed system.

5.6.2. Application performance

Most any application server slows the applications down in the trade-off for a more flexible environment for the components and so does MVCI reference implementation. The indirection mechanism introduced by the component delegate architecture causes some slowdown to the system. The interface translation causes even more overhead, especially with the brute-force interface-to-adapter copying implemented in the MVCI reference implementation.

Table 5. Measured raw method invocation performance of the MVCI reference implementation against direct invocation in Java. In the tests, 0 - 2 interface translations were in use.

<table>
<thead>
<tr>
<th>Invocations/ms</th>
<th>% of Direct invocation</th>
<th>% of v1 -&gt; v1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Invocation</strong></td>
<td>702</td>
<td>100.0 %</td>
</tr>
<tr>
<td>v1 -&gt; v1</td>
<td>520</td>
<td>74.1 %</td>
</tr>
<tr>
<td>v1 -&gt; v2</td>
<td>51</td>
<td>7.3 %</td>
</tr>
<tr>
<td>v1 -&gt; v3</td>
<td>27</td>
<td>3.8 %</td>
</tr>
<tr>
<td>v2 -&gt; v3</td>
<td>52</td>
<td>7.4 %</td>
</tr>
</tbody>
</table>
Table 5 summarizes the raw method invocation performance of the MVCI reference implementation. The raw performance is about 74 percent of the performance of a direct Java method invocation without any translation. With the translations in place, the raw performance heavily degrades due to the computationally-intensive interface translation code. With one translation, the performance is about 7.3 % - 7.4 % of the direct invocation performance and around 9.8 % - 10.0 % of the performance of the component in the MVCI reference implementation without any translations. The raw performance further degrades with two translations to mere 3.8 % of the direct invocation and 5.2 % of the performance of the MVCI component without any translations.

Table 6. Projected MVCI reference implementation performance in percent of direct invocation when the time spent in the actual method is 0.2 - 1.0 milliseconds.

<table>
<thead>
<tr>
<th>Method time</th>
<th>Direct</th>
<th>v1-v1</th>
<th>v1-v3</th>
<th>v1-v5</th>
<th>v1-v7</th>
<th>v1-v9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2 ms</td>
<td>100 %</td>
<td>99.8 %</td>
<td>85.2 %</td>
<td>74.5 %</td>
<td>66.1 %</td>
<td>59.5 %</td>
</tr>
<tr>
<td>0.4 ms</td>
<td>100 %</td>
<td>99.9 %</td>
<td>92.0 %</td>
<td>85.3 %</td>
<td>79.6 %</td>
<td>74.5 %</td>
</tr>
<tr>
<td>0.6 ms</td>
<td>100 %</td>
<td>99.9 %</td>
<td>94.5 %</td>
<td>89.7 %</td>
<td>85.4 %</td>
<td>81.4 %</td>
</tr>
<tr>
<td>0.8 ms</td>
<td>100 %</td>
<td>99.9 %</td>
<td>95.8 %</td>
<td>92.1 %</td>
<td>88.6 %</td>
<td>85.4 %</td>
</tr>
<tr>
<td>1.0 ms</td>
<td>100 %</td>
<td>100 %</td>
<td>96.6 %</td>
<td>93.5 %</td>
<td>90.7 %</td>
<td>88.0 %</td>
</tr>
</tbody>
</table>

The whole picture of performance is not shown by Table 5 as there are other aspects to take into consideration in addition to raw performance. We need to factor in the time spent in the actual method where the component is performing the business logic. Additionally, on distributed systems, the network latency easily increases the method invocation times up to a few milliseconds.

The time spent in the business method execution and the additional latency introduced by a distributed environment is significant compared to the translation overhead for the MVCI reference implementation. From the data in Table 5 we can calculate that the overhead for a translation in the MVCI reference implementation is around 0.0196 milliseconds on the test hardware (test environment details are available in Appendix C). The overhead for two translations is about 0.0370 milliseconds, which is about two times the overhead for a single translation.

Table 6 shows the projected performance of a component in the MVCI reference implementation when the time spent executing the actual method varies between 0.2
and 1.0 milliseconds. With eight translations in sequence between the interface versions v1 and v9, the projected performance is within 59.5 % - 88.0 % of direct invocation depending on the time spent in the method and the invocation overhead (network latency, database access, etc.)

Based on the projected performance presented in Table 6, we argue that the actual performance of the whole system is not significantly affected by the translations introduced by the MVCI reference implementation. Furthermore, a large number of clients would be using the newest version of the interface and thus getting the performance within the range of 74.1 % - 100 % of a direct method invocation. More detailed performance measurements are presented in Appendix C.
6. Evaluation of MVCI

In chapter 3.4 we laid out the requirements for a system capable of dynamic updates. In this chapter we will evaluate how well the MVCI framework presented in chapters 4 and 5 meets these requirements.

The goals introduced in chapters 3.4.1 and 3.4.2, dynamic updates and upgrades was the starting point for this thesis and MVCI fulfills both of these goals. It is possible to update and upgrade components to MVCI without disturbing the system – a fact proved by the MVCI reference implementation. The clients are able to use their old interfaces and there may be multiple clients using different versions of the component interface concurrently as defined in chapters 3.4.3 and 3.4.5. The interface evolution in the MVCI framework is free as required by chapter 3.4.7, but the MVCI reference implementation introduces some limitations. The reference implementation only allows a single interface for a component but that should not be impossible to overcome – it's just a small matter of software engineering in the MVCI implementation area.

All clients are – as required by chapter 3.4.6 – served by a single implementation version that corresponds to the latest version of the component installed in the MVCI reference implementation. The performance of MVCI reference implementation does degrade when more interface translations occur but not significantly, as defined in chapter 3.4.10. The performance overhead of translators is negligible in any real-world system that is not designed to measure the raw method call performance. The development of the component-based applications gets different with MVCI but, as we argue in chapter 5.6.1, it does not significantly complicate the developers' work and thus, MVCI complies with the requirement of chapter 3.4.9.

The MVCI framework only provides a cursory guideline on how to cope with the state transfer of components defined in chapter 3.4.4 and the reference implementation does not support it at all. There still are problems to be solved with the component state transfer in the MVCI framework, especially on how to coordinate the state transfer with multiple concurrent clients accessing multiple versions of the interfaces during a complex dynamic reconfiguration operation.

Most of the data types are addressed by this thesis and by the MVCI framework as required by chapter 3.4.8 but the handling of callbacks and remote object invocations is not solved and would need further development of the framework. The MVCI framework is designed to be programming language independent and the reference implementation proves that it is operating system independent as it works on several
operating systems using the Java Standard Edition [Java SE, 2008] platform in a way aligned with chapter 3.4.11.

In conclusion, the MVCI framework fulfills at least partially all of the goals set in chapter 3.4. There is still work to do to define how the state transfer, the data types, especially the callback type and the programming language independence is realized in an evolutionary, dynamically updatable externally multi-versional component framework.
7. Conclusions

In this thesis we have laid out the requirements for a component framework that is capable of dynamic updates while still supporting the system-internal and system-external clients using an old version of the component interface. This allows truly evolutionary component development that solves many of the problems of coping with the legacy interfaces. We identified five domains – client-side external, client, middleware, server and server-side external – where the requirements can be addressed and evaluated the suitability of each domain for the task.

We selected the server domain for further inspection and presented MVCI – a server domain framework capable of supporting evolutionary component development. The different ways of coping with the interface compatibility problem, where the old versions of the interface must be supported while enabling the component evolution, were identified. The MVCI framework supports the traditional solution where the interface evolution is restricted so that anything that breaks the backward compatibility is forbidden, the simple interface translation solution where each interface version has its own translator that directly translates to the newest version of the interface, and the transitive interface translation where each version of the interface has a translator that only translates to the next version of the interface. It is also possible to combine these methods to gain performance- or other benefits.

MVCI builds on strict separation of component implementation from the component interface – in MVCI even the component version identifier has own version numbers for the interface versions and for the implementation version. This strict separation allows us to introduce new architectural elements that provide a solution to translating a request from an old version of an interface to another version of the interface.

We introduced a version notation to support the interface-implementation separation and multiple versions of multiple interfaces. The notation includes the names and the version numbers of all of the interfaces and the version number of the implementation in the format of \( \{i_1, j_1; \ldots; \} \{i_2, j_2; \ldots; \} \ldots : x \} \) where \( i, j, \ldots \) are interface names, \( i_1, i_2, \ldots, i_n; j_1, j_2, \ldots, j_n; \ldots \) are the version numbers for the corresponding interface name, and \( x \) is the version of the implementation. The notation allows one to identify the state of the system – it is easy to determine which interface versions are supported and what is the implementation version.

We then laid out the architecture for dynamically updating the implementation and upgrading the whole component while still supporting the old versions of the interfaces. The MVCI framework uses interface adapters to overcome the namespace problem that
occurs when there are two different versions of an interface that use identical names. The interface translators in turn translate the component invocation from a version of the interface to another version of the interface. The interface translators can even redirect the invocations to totally different interfaces if required.

The MVCI reference implementation, which is an implementation of the MVCI framework in the Java programming language on the Java platform, was introduced as a proof-of-concept implementation. The MVCI reference implementation is capable of running several versions of interfaces of a single component while only running one component implementation for these interfaces. The dynamic installation, update and upgrade operations are fully supported during ongoing concurrent client connections.

We described the Java class loader hierarchy necessary to implement the MVCI reference implementation. Each component needs to have separate class loaders for at least the component interface, the component implementation and the interface adapter. Additionally, each translator needs its own class loader. This arrangement allows independent evolution of the components by providing namespace separation for the components and by enabling the dynamic updates of the different parts of the components.

The dynamic reconfiguration operations on the MVCI reference implementation include installation, update and upgrade of a component. The operations make heavy use of the component registry that keeps books of all of the class loaders, component delegates, interface versions and the implementations of the components. The component metadata contained by the component manifest file is essential for the reconfiguration operation to work. The MVCI reference implementation can compute the type of the required operation – installation, update or upgrade – by using the component metadata in the component manifest file and in the component registry.

While designing the MVCI reference implementation, we had a good performance as one goal. While the MVCI raw method invocation performance is quite poor when using any translators, the real-world performance, where the business logic execution is assumed to take some time and there is an invocation overhead from for example network latency, is quite acceptable with around 59.5 % - 100 % of direct method invocation performance.

Finally, we evaluated MVCI framework and the reference implementation against the goals we set in the chapter 3.4 of this thesis. The MVCI framework clearly meets most of the goals as only the state transfer to updated component, the support for all
imaginable data types and the programming language independence would need more work on the MVCI framework.
References


Appendices

Appendix A: Sample JAR manifest file for the MVCI reference implementation

Manifest-Version: 2.0
Created-By: Joonas Haapsaari

Name: mvci.component
Component-Name: Component1
Interface-Version: 3
Interface-Jars: c3inf.jar
Interface-Class: fi.uta.joonashaapsaari.compo1.ComponentOne
Implementation-Version: 1
Implementation-Jars: c3impl.jar
Implementation-Class: fi.uta.joonashaapsaari.compo1.impl.COneImpl
Adapter-Jars: c3adapter.jar
Translator-From-Interface-Version: 2
Translator-Jars: c2c3translator.jar
Translator-Class: fi.uta.joonashaapsaari.compoltranslators.TranslatorV2V3
Appendix B: Source code for two versions of a component interface, an adapter and a translator

// Component1 interface version 2
package fi.uta.joonashaapsaari.compo1;
public interface ComponentOne
{
    public void invoke(long key, Payload data) throws PayloadException;
}

package fi.uta.joonashaapsaari.compo1;
public class Payload
{
    private String name;
    private String value;
    private String version;

    public Payload(String name, String value, String version)
    {
        super();
        this.name = name;
        this.value = value;
        this.version = version;
    }

    public Payload()
    {
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getValue()
    {
        return value;
    }

    public void setValue(String value)
    {
        this.value = value;
    }

    public String getVersion()
    {
        return version;
    }

    public void setVersion(String version)
    {
        this.version = version;
    }
}

package fi.uta.joonashaapsaari.compo1;
public class PayloadException extends Exception
{
    public PayloadException(String message)
    {
        super(message);
    }
}

// Component1 interface version 3

package fi.uta.joonashaapsaari.compo1;
public interface ComponentOne
{
    public boolean preinvoke(long key);
    public void postinvoke(Payload data) throws PayloadException;
}

package fi.uta.joonashaapsaari.compo1;
public class Payload
{
    private String name;
    private String value;
    private String version;

    public Payload(String name, String value, String version)
    {
        super();
        this.name = name;
        this.value = value;
        this.version = version;
    }

    public Payload()
    {
    }

    public String getName()
    {
        return name;
    }
}
public void setName(String name)
{
    this.name = name;
}

public String getValue()
{
    return value;
}

public void setValue(String value)
{
    this.value = value;
}

public String getVersion()
{
    return version;
}

public void setVersion(String version)
{
    this.version = version;
}

package fi.uta.joonashaapsaari.compo1;
public class PayloadException extends Exception
{
    public PayloadException(String message)
    {
        super(message);
    }
}

// Adapter for Component1 version 2
package _v2.fi.uta.joonashaapsaari.compo1;
public interface ComponentOne
{

public void invoke(long key, Payload data) throws PayloadException;
}

package _v2.fi.uta.joonashaapsaari.compo1;
public class Payload {
    private String name;
    private String value;
    private String version;

    public Payload(String name, String value, String version) {
        super();
        this.name = name;
        this.value = value;
        this.version = version;
    }

    public Payload() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }
}

package _v2.fi.uta.joonashaapsaari.compo1;
public class PayloadException extends Exception {
    public PayloadException(String message) {
        super(message);
    }
}
package fi.uta.joonashaapsaari.compo1translators;
import fi.uta.joonashaapsaari.compo1.ComponentOne;
import fi.uta.joonashaapsaari.compo1.Payload;
import fi.uta.joonashaapsaari.compo1.PayloadException;
import fi.uta.joonashaapsaari.mvci.translator.AbstractTranslatorBase;

public class TranslatorV2V3 extends AbstractTranslatorBase implements _v2.fi.uta.joonashaapsaari.compo1.ComponentOne
{
    public TranslatorV2V3()
    {
        super();
    }

    public void invoke(long key, _v2.fi.uta.joonashaapsaari.compo1.Payload data) throws _v2.fi.uta.joonashaapsaari.compo1.PayloadException
    {
        Payload newPayload= new Payload();
        newPayload.setName(data.getName());
        newPayload.setValue(data.getValue());

        try
        {
            if (((ComponentOne)target).preinvoke(key) == false)
                throw new _v2.fi.uta.joonashaapsaari.compo1.PayloadException("Preinvoke failed!");

            ((ComponentOne)target).postinvoke(newPayload);
        }
        catch(PayloadException e)
        {
            throw new _v2.fi.uta.joonashaapsaari.compo1.PayloadException(e.getMessage());
        }
        finally
        {
            data.setName(newPayload.getName());
            data.setValue(newPayload.getValue());
        }
    }
}
Appendix C: MVCI reference implementation performance benchmarks

The performance benchmarks presented here were performed on a single IBM ThinkPad T30 laptop with 512 megabytes of RAM. The laptop was running Linux operating system.

The benchmarks were run with only one client connecting to a single server component without any multithreading. The unloaded performance tests in Table vii were run so that twelve rounds of each tests was performed and an average of all tests was taken for Table vii.

The projected performance tests in Table viii, ix and x were computed based on the results of the tests run for Table vii. The average overhead of a translation from version to the next version was computed and it was used as a factor in projecting the performance figures for translation from version 1 to versions larger than 3 (i.e. figures for v1 => v4 – v1 => v10 are computed using the average overhead translation value.

### Unloaded Performance percentage of direct invocation

<table>
<thead>
<tr>
<th>Unloaded Invocation type</th>
<th>V1 =&gt; V1</th>
<th>V1 =&gt; V2</th>
<th>V1 =&gt; V3</th>
<th>V2 =&gt; V2</th>
<th>V2 =&gt; V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>0.74</td>
<td>0.07</td>
<td>0.04</td>
<td>0.74</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table vii. MVCI reference implementation performance difference to direct method invocation in percentage.

### Performance estimates for 0 – 1 ms spent in invoked method

<table>
<thead>
<tr>
<th>t</th>
<th>Direct</th>
<th>V1 =&gt; V1</th>
<th>V1 =&gt; V2</th>
<th>V1 =&gt; V3</th>
<th>V1 =&gt; V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0014235584</td>
<td>0.0019225531</td>
<td>0.0194381157</td>
<td>0.0364636957</td>
<td>0.0534892756</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0104235584</td>
<td>0.0119225531</td>
<td>0.1194381157</td>
<td>0.1364636957</td>
<td>0.1534892756</td>
</tr>
<tr>
<td>0.2</td>
<td>0.0204235584</td>
<td>0.0219225531</td>
<td>0.2194381157</td>
<td>0.2364636957</td>
<td>0.2534892756</td>
</tr>
<tr>
<td>0.3</td>
<td>0.0304235584</td>
<td>0.0319225531</td>
<td>0.3194381157</td>
<td>0.3364636957</td>
<td>0.3534892756</td>
</tr>
<tr>
<td>0.4</td>
<td>0.0404235584</td>
<td>0.0419225531</td>
<td>0.4194381157</td>
<td>0.4364636957</td>
<td>0.4534892756</td>
</tr>
<tr>
<td>0.5</td>
<td>0.0504235584</td>
<td>0.0519225531</td>
<td>0.5194381157</td>
<td>0.5364636957</td>
<td>0.5534892756</td>
</tr>
<tr>
<td>0.6</td>
<td>0.0604235584</td>
<td>0.0619225531</td>
<td>0.6194381157</td>
<td>0.6364636957</td>
<td>0.6534892756</td>
</tr>
<tr>
<td>0.7</td>
<td>0.0704235584</td>
<td>0.0719225531</td>
<td>0.7194381157</td>
<td>0.7364636957</td>
<td>0.7534892756</td>
</tr>
<tr>
<td>0.8</td>
<td>0.0804235584</td>
<td>0.0819225531</td>
<td>0.8194381157</td>
<td>0.8364636957</td>
<td>0.8534892756</td>
</tr>
<tr>
<td>0.9</td>
<td>0.0904235584</td>
<td>0.0919225531</td>
<td>0.9194381157</td>
<td>0.9364636957</td>
<td>0.9534892756</td>
</tr>
<tr>
<td>1</td>
<td>1.0004235584</td>
<td>1.019225531</td>
<td>1.0194381157</td>
<td>1.0364636957</td>
<td>1.0534892756</td>
</tr>
</tbody>
</table>

Table viii. Projected performance estimates in milliseconds/invocation with 0 - 1 milliseconds spend in the invoked method. Table shows the direct invocation time and the time with MVCI reference implementation when there is 0 - 3 translators chained for the invocation.
### Performance estimates for 0 – 1 ms spent in invoked method

<table>
<thead>
<tr>
<th>t</th>
<th>V1 =&gt; V5</th>
<th>V1 =&gt; V6</th>
<th>V1 =&gt; V7</th>
<th>V1 =&gt; V8</th>
<th>V1 =&gt; V9</th>
<th>V1 =&gt; V10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0705148555</td>
<td>0.0875404355</td>
<td>0.1045660154</td>
<td>0.1215915954</td>
<td>0.1386171753</td>
<td>0.1556427553</td>
</tr>
<tr>
<td>0,1</td>
<td>0.1705148555</td>
<td>0.1875404355</td>
<td>0.2045660154</td>
<td>0.2215915954</td>
<td>0.2386171753</td>
<td>0.2556427553</td>
</tr>
<tr>
<td>0,2</td>
<td>0.2705148555</td>
<td>0.2875404355</td>
<td>0.3045660154</td>
<td>0.3215915954</td>
<td>0.3386171753</td>
<td>0.3556427553</td>
</tr>
<tr>
<td>0,3</td>
<td>0.3705148555</td>
<td>0.3875404355</td>
<td>0.4045660154</td>
<td>0.4215915954</td>
<td>0.4386171753</td>
<td>0.4556427553</td>
</tr>
<tr>
<td>0,4</td>
<td>0.4705148555</td>
<td>0.4875404355</td>
<td>0.5045660154</td>
<td>0.5215915954</td>
<td>0.5386171753</td>
<td>0.5556427553</td>
</tr>
<tr>
<td>0,5</td>
<td>0.5705148555</td>
<td>0.5875404355</td>
<td>0.6045660154</td>
<td>0.6215915954</td>
<td>0.6386171753</td>
<td>0.6556427553</td>
</tr>
<tr>
<td>0,6</td>
<td>0.6705148555</td>
<td>0.6875404355</td>
<td>0.7045660154</td>
<td>0.7215915954</td>
<td>0.7386171753</td>
<td>0.7556427553</td>
</tr>
<tr>
<td>0,7</td>
<td>0.7705148555</td>
<td>0.7875404355</td>
<td>0.8045660154</td>
<td>0.8215915954</td>
<td>0.8386171753</td>
<td>0.8556427553</td>
</tr>
<tr>
<td>0,8</td>
<td>0.8705148555</td>
<td>0.8875404355</td>
<td>0.9045660154</td>
<td>0.9215915954</td>
<td>0.9386171753</td>
<td>0.9556427553</td>
</tr>
<tr>
<td>0,9</td>
<td>0.9705148555</td>
<td>0.9875404355</td>
<td>1.0045660154</td>
<td>1.0215915954</td>
<td>1.0386171753</td>
<td>1.0556427553</td>
</tr>
<tr>
<td>1</td>
<td>1.0705148555</td>
<td>1.0875404355</td>
<td>1.1045660154</td>
<td>1.1215915954</td>
<td>1.1386171753</td>
<td>1.1556427553</td>
</tr>
</tbody>
</table>

Table ix. Projected performance estimates in milliseconds/invocation with 0 - 1 milliseconds spend in the invoked method. Table shows the time with MVCI reference implementation when there is 4 - 9 translators chained for the invocation.

### Estimates for performance percentage of direct invocation with 0 – 1 ms spent in invoked method

<table>
<thead>
<tr>
<th>%</th>
<th>Direct</th>
<th>V1V1</th>
<th>V1V2</th>
<th>V1V3</th>
<th>V1V4</th>
<th>V1V5</th>
<th>V1V6</th>
<th>V1V7</th>
<th>V1V8</th>
<th>V1V9</th>
<th>V1V10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.000</td>
<td>0.740</td>
<td>0.073</td>
<td>0.039</td>
<td>0.027</td>
<td>0.020</td>
<td>0.016</td>
<td>0.014</td>
<td>0.012</td>
<td>0.010</td>
<td>0.009</td>
</tr>
<tr>
<td>0,1</td>
<td>1.000</td>
<td>0.995</td>
<td>0.849</td>
<td>0.743</td>
<td>0.661</td>
<td>0.595</td>
<td>0.541</td>
<td>0.496</td>
<td>0.458</td>
<td>0.425</td>
<td>0.397</td>
</tr>
<tr>
<td>0,2</td>
<td>1.000</td>
<td>0.998</td>
<td>0.918</td>
<td>0.852</td>
<td>0.795</td>
<td>0.745</td>
<td>0.701</td>
<td>0.661</td>
<td>0.626</td>
<td>0.595</td>
<td>0.566</td>
</tr>
<tr>
<td>0,3</td>
<td>1.000</td>
<td>0.998</td>
<td>0.944</td>
<td>0.896</td>
<td>0.853</td>
<td>0.814</td>
<td>0.778</td>
<td>0.745</td>
<td>0.715</td>
<td>0.687</td>
<td>0.662</td>
</tr>
<tr>
<td>0,4</td>
<td>1.000</td>
<td>0.999</td>
<td>0.957</td>
<td>0.920</td>
<td>0.885</td>
<td>0.853</td>
<td>0.823</td>
<td>0.796</td>
<td>0.770</td>
<td>0.745</td>
<td>0.722</td>
</tr>
<tr>
<td>0,5</td>
<td>1.000</td>
<td>0.999</td>
<td>0.965</td>
<td>0.935</td>
<td>0.906</td>
<td>0.879</td>
<td>0.853</td>
<td>0.829</td>
<td>0.807</td>
<td>0.785</td>
<td>0.765</td>
</tr>
<tr>
<td>0,6</td>
<td>1.000</td>
<td>0.999</td>
<td>0.971</td>
<td>0.945</td>
<td>0.920</td>
<td>0.897</td>
<td>0.875</td>
<td>0.854</td>
<td>0.833</td>
<td>0.814</td>
<td>0.796</td>
</tr>
<tr>
<td>0,7</td>
<td>1.000</td>
<td>0.999</td>
<td>0.975</td>
<td>0.952</td>
<td>0.931</td>
<td>0.910</td>
<td>0.891</td>
<td>0.872</td>
<td>0.854</td>
<td>0.836</td>
<td>0.820</td>
</tr>
<tr>
<td>0,8</td>
<td>1.000</td>
<td>0.999</td>
<td>0.978</td>
<td>0.958</td>
<td>0.939</td>
<td>0.921</td>
<td>0.903</td>
<td>0.886</td>
<td>0.870</td>
<td>0.854</td>
<td>0.839</td>
</tr>
<tr>
<td>0,9</td>
<td>1.000</td>
<td>0.999</td>
<td>0.980</td>
<td>0.963</td>
<td>0.945</td>
<td>0.929</td>
<td>0.913</td>
<td>0.897</td>
<td>0.882</td>
<td>0.868</td>
<td>0.854</td>
</tr>
<tr>
<td>1</td>
<td>1.000</td>
<td>1.000</td>
<td>0.982</td>
<td>0.966</td>
<td>0.951</td>
<td>0.935</td>
<td>0.921</td>
<td>0.907</td>
<td>0.893</td>
<td>0.880</td>
<td>0.867</td>
</tr>
</tbody>
</table>

Table x. Projected performance in percentage of direct invocation with 0 - 1 milliseconds spend in the invoked method. Table shows the direct invocation percentage and the percentage with MVCI reference implementation when there is 0 - 9 translators chained for the invocation.
Appendix D: MVCI source code licensing terms
Multi-Version Component Infrastructure (MVCI)
Copyright (C) 2004-2007 Joonas Haapsaari
joonas (dot) haapsaari (at) gmail (dot) com

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
Appendix E: GNU General Public License, version 2

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users. This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it. (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.) You can apply it to
your programs, too.

When we speak of free software, we are referring to freedom, not
price. Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have. You must make sure that they, too, receive or can get the
source code. And you must show them these terms so they know their
rights.

We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software. If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

Finally, any free program is threatened constantly by software
patents. We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary. To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and
modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License. The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language. (Hereinafter, translation is included without limitation in
the term "modification".) Each licensee is addressed as "you".


Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separated works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code.  (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it.  For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable.  However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License.  Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

5. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Program or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all.  For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.
To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.
Appendix F: MVCI reference implementation quick guide

The source code of MVCI Framework is included in Appendix G in a special format. It is a BASE64 encoded, BZIP2 compressed TAR archive. To unpack to source code, one needs the following tools

1) BASE64 decoder

2) BUNZIP2 decompressor

3) TAR utility for extracting the TAR archive

The steps for extracting the source code are following:

1) Copy the packaged source code below to a new text file called 'mvci_src.txt'. Ensure that there is nothing else but the source code in the text file. One can achieve this by selecting the packaged source code in the electronic format of this document (PDF) and then copying and pasting the selection to a new text file by the means provided by the operating system.

2) Invoke the BASE64 decoder to the text file created in the previous step ('mvci_src.txt'). Direct the output of the BASE64 decoder to a file called 'mvci_src.tar.bz2'.

3) Use the BUNZIP2 decompressor to the BASE64-decoded file ('mvci_src.tar.bz2'). You should get a new file called 'mvci_src.tar'.

4) Extract the source code from the 'mvci_src.tar' file using the TAR utility. This creates a folder called 'mvci_framework_src' which contains the full source code for MVCI, four versions of a sample MVCI component, two test clients, build files and licensing terms.

To build the MVCI framework, one needs to have Java 2 Standard Edition Runtime (JRE) or Software Development Kit (SDK), version 1.3 or newer (available at http://java.sun.com); and Apache ANT build tool (available at http://ant.apache.org), version 1.6.5 is tested to work but other versions are likely to work too. Once you have these tools installed and configured, you may build the MVCI by invoking:

\texttt{ant all}

To run the MVCI, you may invoke:
This first builds the MVCI if it is not already built and then runs it.

Illustration i. MVCI reference implementation user interface.

Illustration i shows the graphical user interface of the MVCI reference implementation. On the top, one can install, update and upgrade components to a running MVCI implementation. The bottom part allows starting a client application.

There are three versions (1.0, 1.1, 2.0 and 3.0) of a single sample component included with the reference implementation. These can be found in

mvci_framework_src/jars

folder with the name compo1_<version>.jar. Upgrade/update only works from older version to a newer (e.g. from compo1_1.jar to compo1_1_1.jar to compo1_2.jar). The activate button performs the installation, update or upgrade and starts the component. An entry informing on the update is logged to the Ant console.

The client applications can also be found in

mvci_framework_src/jars

with the names clientv1.jar and clientv2.jar. The prior is using the component interface version 1 and the latter is using the interface version 2. There is no client using the version 3 even though the compo1_3.jar implements the version 3 but the translation layer is automatically used to translate from older version to newer one provided that the server components are installed in correct sequence (v1, v2 and
Every click of the Execute button in GUI will start a new instance of the client in the Client Jar file -textbox (see Illustration i) and one can run many clients concurrently, both same and different versions of the client.

Easy way to test the MVCI reference implementation is to first load the component version 1.0 (compo1_1.jar) and then start the client v1 (clientv1.jar). The client executes for quite a long time during which the component can be updated to version 1.1 (compo1_1_1.jar), upgraded to version 2.0 (compo1_2.jar) and to 3.0 (compo1_3.jar). Once the component version 2.0 is installed the client v2 (clientv2.jar) can be started. The clients run happily concurrently both accessing the same component implementation but through different interface translation layer structure.
Appendix G: MVCI reference implementation source code in base64 encoded tar.gz file