KC++ – A Concurrent C++ Programming System

Matti Rintala

Software Systems Laboratory
Tampere University of Technology
ABSTRACT

Abstract

Object-oriented programming and concurrent programming are both becoming more and more common. During the last decade, there has been a lot of research aiming to combine the two, either in the form of a new programming language, a class library, or an extension to an existing language.

This thesis presents KC++ — a concurrent object-oriented programming system. The KC++ system is based on the C++ language, and it introduces concurrency using active objects, asynchronous method calls, and futures. It does this without extending the syntax of C++, adding concurrent semantics to existing C++ concepts. The KC++ system consists of a KC++-to-C++ compiler and a class library.

The main design goal of the system is to make programming with active objects as close as possible to programming with normal C++ objects, without restrictions on how active objects can be created or used. Most existing C++ programming styles, idioms and design patterns should be directly usable in KC++. Existing C++ style analysers and debugging tools can be used with KC++ without modifications.
Preface

The ideas behind this thesis were formed during my stay at King’s College London during 1998. Most of all I want to thank my supervisor Prof. Russel Winder, whose suggestions and knowledge on concurrent C++-based programming helped me enormously. I’d also like to thank everybody else at King’s for making me feel at home there.

I thank Prof. Kai Koskimies for tutoring me after I returned to Tampere University of Technology, and for reviewing the thesis. Likewise I thank Juha Vihavainen from Helsinki University for his help as the second reviewer and his suggestions for improvements.

Thanks are also in order to Jyke and everyone else at the Software Systems Lab, especially for their general patience and answers to my trivial problems.

But most importantly I want to thank Emma for her support and comfort on those dark hours after midnight, when the whole project seemed like a Torment™ [Interplay, 1999]
Contents

1 Introduction
  1.1 Need for concurrency ........................................... 1
  1.2 A language extension vs. a library ............................. 2
  1.3 Active objects .................................................. 3
  1.4 Motivation behind KC++ .......................................... 4
  1.5 The history of KC++ .............................................. 5
  1.6 The structure of this thesis ..................................... 5

2 Overview of KC++
  2.1 The KC++ active object concept ................................ 7
    2.1.1 Active classes ............................................... 7
    2.1.2 Creating active objects .................................... 8
    2.1.3 Proxies and reference counting .......................... 9
  2.2 Futures .......................................................... 11
    2.2.1 Normal futures .............................................. 11
    2.2.2 void-futures ............................................... 12
    2.2.3 Future sources .............................................. 13
  2.3 Passing parameters to active objects .......................... 13
    2.3.1 Normal parameters .......................................... 14
    2.3.2 Future parameters .......................................... 14
    2.3.3 Future reference parameters .............................. 14
    2.3.4 Active object parameters ................................. 15
  2.4 Locking active objects ......................................... 15

3 Active objects and proxies ........................................ 17
  3.1 Making objects active ......................................... 17
## CONTENTS

3.2 Proxy classes .................................................. 18
   3.2.1 Internals of a proxy ....................................... 19
   3.2.2 Lifetime of an active object .............................. 20
   3.2.3 Proxies and inheritance .................................... 21
3.3 Inheritance and active classes ................................. 22
3.4 Calling active object methods ................................ 23
   3.4.1 Local method calls ........................................ 24
   3.4.2 Accessing private members of other objects ............. 25
3.5 Choosing the correct method .................................. 26
   3.5.1 Basic types ................................................ 27
   3.5.2 Class types and structs ................................... 28
   3.5.3 Pointer and reference types ............................... 28
3.6 Returning values from active objects ......................... 30

4 Passing parameters to active objects .......................... 32
   4.1 Marshalling and unmarshalling .............................. 32
   4.2 Different types of parameters ............................... 33
      4.2.1 Basic types ............................................. 33
      4.2.2 Non-active user-defined classes and structs ........... 33
      4.2.3 Enumeration types and non-user-defined types ......... 34
      4.2.4 Active classes .......................................... 35
      4.2.5 Pointers and references to active objects ............. 35
      4.2.6 Pointers and references to non-active types .......... 36
      4.2.7 Futures and references to futures ..................... 36
   4.3 Pointers and references to active objects .................. 37
      4.3.1 References to active objects .......................... 38
      4.3.2 Pointers to active objects ............................ 42
   4.4 Passing active objects by value ............................ 44

5 Asynchronous method calls and futures ......................... 47
   5.1 Principles of futures in KC++ ............................... 48
      5.1.1 The interface of futures ............................... 49
      5.1.2 Using futures ........................................... 50
      5.1.3 Futures as constant references to “system-wide” data 53
## CONTENTS

5.1.4 Creating futures ........................................... 55  
5.1.5 Assigning futures ......................................... 56  
5.1.6 Destroying futures ........................................ 57  
5.2 Futures as return values ..................................... 58  
5.2.1 Asynchronous method calls ................................. 62  
5.2.2 Synchronous method calls .................................. 64  
5.2.3 Explicit future return types ............................... 64  
5.3 Futures as parameters ....................................... 66  
5.3.1 Futures as value parameters ............................... 67  
5.3.2 Futures as reference parameters ......................... 68  
5.4 Pure synchronisation — void-futures ....................... 69  
5.5 Creating futures explicitly — future sources ............. 71  
5.6 Locking active objects ...................................... 76  

6 Implementation of KC++ ........................................ 79  
6.1 The KC++ compiler .......................................... 79  
6.1.1 Active class definitions .................................. 81  
6.1.2 Uses of active objects, pointers and references ...... 82  
6.1.3 Additional changes to pointers ........................... 84  
6.2 The KC++ library ........................................... 84  
6.2.1 Implementation of the base class Active ............... 85  
6.2.2 Implementation of proxies, pointers and references .. 85  
6.2.3 Implementation of futures ............................... 87  
6.2.4 Implementation of message streams ................. 90  
6.2.5 Implementation of unmarshalling ...................... 91  

7 Related work .................................................... 94  
7.1 Classification of concurrent object-oriented systems .... 94  
7.2 Concurrent object-oriented languages ..................... 95  
7.3 Concurrent object-oriented libraries ....................... 96  
7.4 Concurrent extensions to C++ .............................. 97  

8 Conclusions and future work ................................... 100  
8.1 Advantages of KC++ ....................................... 100  
8.1.1 Familiar syntax and “feel” ............................. 100
# Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The $\text{KC}+$ proxy reference counting model</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Different ways of making a class active in $\text{KC}+$</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Inheritance hierarchies of active classes and proxies</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>An active object method call</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Creating the method message</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Passing a polymorphic object</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Returning a value from an active object method</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Passing an active object pointer between active objects</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Passing an active object by value</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Different ways to implement futures</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Destruction of a future after receiving the value</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>Destruction of a future before receiving the value</td>
<td>60</td>
</tr>
<tr>
<td>6.1</td>
<td>The $\text{KC}+$ compilation process</td>
<td>80</td>
</tr>
<tr>
<td>6.2</td>
<td>The $\text{KC}+$ future implementation</td>
<td>88</td>
</tr>
</tbody>
</table>
Code listings

2.1 Creation of an active class and active objects .................... 8
2.2 Part of code generated from the code in Listing 2.1 ................. 10

4.1 Marshalling/unmarshalling mechanisms of a class ................ 35
4.2 Marshalling/unmarshalling of an enumeration ..................... 36
4.3 Object references and asynchronous calls .......................... 41

5.1 Example where int has been replaced with futures ............... 52
5.2 Definitions of an active class and its proxy class ............... 61
5.3 Futures with ISO C++ containers .................................. 63
5.4 Use of explicit future return types ................................ 66
5.5 Listing 5.4 with a non-future return type ......................... 67
5.6 Synchronisation with void-futures ................................ 70
5.7 Outline of an active class implementing a barrier ............... 72
5.8 The barrier class using a future source ........................... 74
5.9 A reusable barrier class ............................................ 75
5.10 Object locking using lock objects ................................. 77
5.11 Lock objects as lock tokens with auto_ptr ....................... 78

6.1 Compiled active class definitions ................................. 83
6.2 Changes to the this pointer ........................................ 85
6.3 A compiler-generated proxy method ................................. 86
6.4 The definition of SPECIAL_UNMARSHAL ......................... 93
Chapter 1

Introduction

During the last decade, object-oriented programming has ceased to be just “the latest fad” and has established its position among the respected mainstream software design paradigms. Its benefits include strong and intuitive encapsulation (if used correctly) with classes, and code reuse and limited generic programming through inheritance and dynamic binding. It has proven its usefulness, even though it has also demonstrated that writing reusable code requires careful planning and extra work, even with object-oriented programming.

At the moment the most widely used language for object-oriented programming is C++. Because of its historical background, suitability for system-level programming and fame as “an efficient language,” it has become a “default language” in many companies and universities. C++ has gained its position despite the fact that strictly speaking it is not a pure object-oriented programming language, but a language “that supports Object-Oriented and other useful styles of programming” [Stroustrup, 2000].

1.1 Need for concurrency

While object-oriented programming has established its position in software design, need for concurrency in computer programs has increased rapidly. This need arises from several directions. One obvious way to increase computing performance is to use several processors in parallel. Current computer networks and “global computing” have created a need for distributed programs, where different parts of the program reside in different computers and possibly execute in parallel. Modern graphical user interfaces and other reactive systems often require that a program responds to outside events seemingly concurrently, even in a one-processor environment.

For these reasons research on concurrent and parallel programming has been active at the same time as object-oriented programming has become popular. It was inevitable that the idea to combine these two would arise quite soon. However, most object-oriented programming languages were not originally designed with concur-
CHAPTER 1. INTRODUCTION

rency in mind, so many different approaches have been tried to introduce concurrency into the world of object-oriented programming (see [Agha et al., 1993], for example).

1.2 A language extension vs. a library

Clearly one way to combine object-oriented programming and concurrency is to create a whole new programming language designed just for this purpose. However, it is very difficult to get people to use a completely new programming language in today’s world where even many existing small languages face extinction, i.e. to become languages “which are only of academic interest.” Most software projects rely heavily on existing libraries written in some existing programming language, and using such libraries from another programming language would often be painful. Programming support tools like user-interface generators, visualisation tools and debuggers are also language-specific, and would have to be rewritten for a new language.

For the reasons above, most concurrent programming systems are based on an existing language. The chosen language and its features then dictate how concurrency can be added to the language. The subject of this thesis, KC++ (King’s College C++), is based on the C++ language, mainly because of popularity and efficiency C++ (and the fact that its predecessor UC++ was also based on C++).

There are several ways to add support for concurrency to a programming language like C++, and many such additions are presented in [Wilson and Lu, 1996]. One way is to support concurrency by providing a library, so users do not have to learn new language features to be able to benefit from concurrency. Many such libraries exist for C++, for example ABC++ [O’Farrell et al., 1996], The Amelia Vector Template Library [Sheffler, 1996], and CHAOS++ [Chang et al., 1996].

A problem with such libraries is that since the underlying programming language does not support concurrency, the syntax of the library can be awkward and non-intuitive. This is especially true if the paradigm behind the library does not resemble the paradigm behind the language. It may also make learning to use the library quite difficult, as knowing the underlying language does not necessarily help learning the library.

A second option is to extend an existing programming language by adding new keywords and constructs necessary for concurrent programming. MPC++ [Ishikawa et al., 1996], ICC++ [Chien and Dolby, 1996], and UC++ [Winder et al., 1996] are examples of such extensions to C++.

From the programmers viewpoint language extensions are a satisfactory solution, as most of the language stays the same and the programmer only has to learn the new constructs. The implementation is also often quite portable and efficient, as it is possible to use a precompiler to map the new constructs to existing programming language constructs and calls to a run-time library providing support for concurrency.
However, extensions to language syntax cause problems with automatic style checkers, code generators, and other tools which are written to accept a certain language syntax. Debugging programs may also become difficult if new constructs do not map directly to existing language constructs.

### 1.3 Active objects

When concurrency is introduced into object-oriented programming, it is quite natural to attempt to merge concurrency and objects — after all, object-oriented programming is about thinking about everything in terms of objects. This leads to the concept of *active objects*. Active objects are objects whose methods are executed concurrently with the rest of the program. The idea of active objects is common enough to be classified as a design pattern [Lavender and Schmidt, 1995].

Object level is not suitable for very low-level concurrency. For example, it is not reasonable to model individual machine-code instructions etc. as “objects” and try to execute them in parallel. However, it is quite usable in higher-level concurrency. Objects are encapsulated from the rest of the program, and this encapsulation makes it possible to put the internals of an object in its own address space — another processor with its own memory or even a remote machine. At the same time the encapsulation enforces the idea of strict interfaces which are the only way to use an object. These interfaces provide a natural way of communicating between two concurrently running parts of a program. It should be noted, however, that the active object approach is not the only one possible, but concurrency can of course be introduced as a mechanism completely independent of objects. However, these approaches are outside the scope of this thesis.

If active objects are chosen as the way to introduce concurrency to object-oriented programming, there is still the decision of whether two or more methods of one active object are allowed to execute concurrently with each other. There are benefits in both possibilities. Allowing several concurrent methods is more flexible, but it also involves difficult mutual exclusion and synchronisation problems. If the number of simultaneously running methods is restricted to one for each active object, there are no internal mutual exclusion problems, as only one method may access the internals of an active object at any time. These kinds of active objects resemble *monitors*, a programming construct to handle mutual exclusion in concurrent programming, made widely known by Hoare [Hoare, 1974]. The KC+ system described in this thesis takes the latter choice and allows only one running method per active object.

Even with just one method at a time, there is still one choice to be made: whether to allow a method to give way to another method and continue afterwards. This behaviour corresponds to the “signal” and “wait” operations in monitors. On the other hand this “voluntary interruption” is sometimes quite handy, but on the other hand it requires activies objects to remember and store the state of the method execution, execute another method and then use the stored state to continue the execution of
the original method. This can be implemented using multiple threads in each active object, but it increases the overhead of a method call. The current KC++ system follows the path of its predecessor, UC++ (see Section 1.5), and does not allow methods to be interrupted. This makes active objects fast and easy to implement, but also restricts their use (Chapter 8 lists future plans for KC++, including support for method interruption).

1.4 Motivation behind KC++

The KC++ system tries to overcome the problems mentioned in Section 1.2 by providing new concurrent semantics to some existing C++ constructs. The system consists of a KC++-to-C++ compiler and a class library which provides run-time support for concurrent processes and their communication. KC++ programs are syntactically valid C++ programs (and vice versa), so existing C++ style analysers, statistical tools etc. can be used with KC++ programs without modifications. In this respect KC++ resembles C++// [Caromel et al., 1996], which also introduces concurrency without extending the C++ syntax.

KC++ is based on the UC++ language [Winder et al., 1996] and many of its design goals are similar to UC++. It tries to add concurrency support to C++ as “naturally” as possible, allowing existing object oriented language features and C++ programming idioms to be used, without forcing the programmer to use of any particular coding and design style.

Like in UC++, concurrency in KC++ is based on the active object concept. A concurrent KC++ program usually consists of several active objects communicating with each other. The active object model is also behind many other concurrent C++ systems, for example C++// and ABC++.

Exceptions are now becoming more widely used in C++, and exception safety\(^1\) is rapidly becoming more and more important in C++ programming (see [Cargill, 1994], [Reeves, 1996], [Sutter, 1997], and [Sutter, 2000], for example). KC++ tries to support programming idioms needed in exception-safe C++ programming [Austern, 1998]. For example, KC++ active objects can be created as local variables, allowing programmers to rely on C++ stack unwinding to destroy unnecessary objects when an exception is thrown.

The code produced by the KC++ compiler is close to the original KC++ code. Most modifications are only changes to type names. This means that the code produced by the compiler is expected to be debuggable with most normal C++ debuggers.

\(^1\)The term exception safe is usually used to describe a part of a program which continues to work properly no matter what exceptions occur.
CHAPTER 1. INTRODUCTION

1.5 The history of KC++

The history of the KC++ system began in 1998 during my half-year stay in the King's College London under the supervision of Prof. Russel Winder. He had earlier been working on UC++, another C++-based concurrent programming system. The UC++ system is also based on the idea of active objects with one-method-run-to-completion semantics. When I arrived to King’s College looking for a research subject, Prof. Winder came up with the idea to start from the ideas behind UC++ and design a new system which would fit “more nicely” into the C++ language (Chapter 7 mentions some problematic features in UC++).

During my stay I studied some existing concurrent programming systems based on the C++ language, many of which are presented in “Parallel Programming Using C++” [Wilson and Lu, 1996]. I noticed that most of these systems limited the use of C++ or the way the in which concurrency features could be combined with C++. I then started to design KC++, trying to make it “comfortable” for a C++ programmer and as compatible with the existing C++ language semantics as possible. Frequent discussions with Prof. Winder helped me to refine some ideas and produced even more ideas for the KC++ system.

After the KC++ system started to take shape, Prof. Winder acquired a licence to use a C++ compiler front end from the Edison Design Group (EDG) [EDG, 1999]. During the latter part of my stay I wrote a test implementation of the KC++ compiler and the KC++ library. The fact that KC++ doesn’t extend the C++ syntax helped me enormously, because I only had to write a back end to the EDG front end.

After I returned to Finland I did not work on KC++ for some time, but returned to it in the spring 1999, when I wrote an article on KC++ (still unpublished), which is the basis of Chapter 2. I then made some minor improvements to the compiler and started to write this thesis (which was interrupted by another writing project, and took a year to get finished).

1.6 The structure of this thesis

This thesis describes the KC++ system — a concurrent programming environment based on the C++ language and active objects. It contains both the description of the KC++ language and the basics of the implementation of the KC++ compiler and runtime library. The next chapters contain the following:

- Chapter 2 gives an overview of the KC++ system without going into too much detail. Its purpose is to give the reader a general idea of the basic structure of KC++. These ideas are then covered in detail in the following chapters.
- Chapter 3 discusses the active objects themselves and compiler-generated proxy classes which the active objects use to communicate with each other.
• Chapter 4 explains how different types of parameters are passed between active objects. This subject is a chapter of its own because active objects do not necessarily have a shared address space, so the standard C++ parameter passing mechanisms cannot be used.

• Chapter 5 shows how asynchronous method calls are implemented in KC++ using future types. It also explains explicit synchronisation with future sources.

• Chapter 6 contains an overview of the KC++ compiler, which is used to compile KC++ programs into concurrent C++ programs. It also explains some main features of the KC++ library, which contains support code needed by KC++.

• Chapter 7 mentions some related concurrent programming systems and briefly compares them to KC++.

• Finally, Chapter 8 gives the concluding words and discusses some ideas for the future versions of KC++.
Chapter 2

Overview of KC++

This chapter gives an overview of the KC++ system. It deliberately does not explain everything in detail, but it is meant to give a picture of the scope of KC++. Details about the features described here are given in the following chapters.

2.1 The KC++ active object concept

The KC++ active object model is in many ways similar to the UC++ active object model. Each active object consists of a thread of execution and a data address space. The data address space contains all the data members of the object, method parameters and any data or objects which have been dynamically created inside the object. Because the address space of each active object may be distinct, active objects may not refer to any global variables or static data members (which are basically global variables).

As mentioned above, each active object in KC++ contains one thread of execution. This thread receives method requests and serves them one at a time. Other method requests are collected in a queue while one method is executing. The execution of a method is non-interruptible (i.e. it follows RTC (run-to-completion) semantics), so if the execution of a method is paused by synchronisation, the whole active object is “blocked” until the method can resume its execution. Usually method requests are served in a strict FIFO (First-In-First-Out) order, but KC++ provides also lock objects (Section 2.4, explained in detail in Section 5.6 in Chapter 5), which allow a user to request an exclusive access to an active object.

2.1.1 Active classes

In KC++ the programmer marks objects as active by deriving their class (directly or indirectly through other base classes) from a base class Active, which is defined in the KC++ library. All objects created from these active classes are automatically active objects. This is similar to the approach used in ABC++ and other “type based” active
CHAPTER 2. OVERVIEW OF KC++

object systems, but different from UC++ where the way the object is created defines whether an object becomes active or not.

2.1.2 Creating active objects

Creation of active objects happens exactly in the same way as creating normal C++ objects. They can be created dynamically with new, as local variables inside a function (or any other code block), as data members of an object (or active object) or even as value parameters to a function or a member function (although this is rarely useful). This is different from most other parallel C++ systems where active objects usually have to be created dynamically with new or in some language specific way. Listing 2.1 shows how active objects can be created in KC++.

The possibility to create active objects without new is essential for exception safety. Objects created with new are not automatically destroyed, so it is the responsibility of

```cpp
class MyAClass : public Active
{
 public:
  explicit MyAClass(int i);
  MyAClass(const MyAClass& r);
  ~MyAClass();
  int foo1(int i);
  MyAClass* foo2(MyAClass& a);
  MyAClass& operator=(const MyAClass& r);
 private:
  // Private implementation details here
};

void func(MyAClass a) // A parameter active object
{
  a.foo1(3);
}

int main()
{
  MyAClass a1(1); // A local active object
  int i1 = a1.foo1(3); // Synchronous
  Future<int> i2 = a1.foo1(4); // Asynchronous
  MyAClass* a2p = new MyAClass(2); // A dynamic active object
  MyAClass* a3p = a1.foo2(*a2p);
  func(a1);
}
```

Listing 2.1: Creation of an active class and active objects
the programmer to make sure that every object (including active objects) is destroyed properly even when exceptions occur. On the other hand, the C++ exception mechanism takes care of the destruction of any object whose lifetime ends because of an exception. This also holds for KC++ active objects, so all active objects which go out of scope because of an exception are automatically destroyed.

2.1.3 Proxies and reference counting

Although active objects are used in KC++ like normal objects, the KC++ compiler has to create special code for them in the resulting C++ code to create new threads of execution, possibly in a different address space. In the rest of this section, the term server refers to the actual concurrently running active object process, and the term client refers to the code which uses the active object.

KC++ uses the Proxy pattern [Gamma et al., 1996, p. 207] to allow the client code to use an active object as if it existed in the client’s own address space. This same technique is used in different forms in many other parallel C++ systems, for example in UC++ and C++//. It is also commonly used in CORBA [OMG, 1998a, Ch. 20].

For each active class, the KC++ compiler creates an internal proxy class and replaces every occurrence of the active class in the client code with this proxy class. The proxy class has the same public interface as the active class, so proxy objects (proxies) can be used in place of the original objects. When a member function of a proxy is called, it forwards the request to the actual active object. This is similar to the “Remote Proxy” pattern in [Rohnert, 1996]. Listing 2.2 on the next page shows parts of the code generated from the code in Listing 2.1.

Many parallel C++ languages introduce proxy classes by deriving them from the active classes. However, this means that either the proxies inherit all the data members of the original classes making the proxies heavy-weight, or proxies are constructed in some way “in place” of the original objects, which results in a compiler-dependent and often non-portable implementation (and usually only works when objects are created with new).

KC++ solves these problems by duplicating the public interface of the active class in the proxy class instead of using inheritance. This way the proxies can be made very light-weight. Avoiding inheritance also allows the KC++ proxy methods to return slightly different return values (futures) to take care of asynchronous return value passing (Section 2.2, explained in detail in Chapter 5).

To keep track of proxies, KC++ implements reference counting in active objects. Each active objects contains a counter which tells how many other active objects contain a proxy referring to the object. Additionally, inside each active object all proxies referring to the same active object share a proxy implementation data structure which has its own local reference count. Figure 2.1 on page 11 shows both reference counters.
 CHAPTER 2. OVERVIEW OF KC++  

1 // In reality proxy classes are defined in separate headers  
2 class Proxy_MyAClass : public Proxy  
3 {  
4     public:  
5         explicit Proxy_MyAClass(int);  
6         // Base class destructor takes care of proxy destruction  
7         MyAClass(const Ref< Proxy_MyAClass > & r);  
8         Future<int> foo1(int);  
9         Ptr< Proxy_MyAClass > foo2(Ref< Proxy_MyAClass > &);  
10         Ref< Proxy_MyAClass > operator =(const Ref< Proxy_MyAClass > & r);  
11         // Special interface for proxy implementation omitted  
12     };  
13     void func(Proxy_MyAClass a) // A parameter active object  
14     {  
15         a.foo1(3);  
16     }  
17     int main()  
18     {  
19         Proxy_MyAClass a1(1); // A local active object  
20         int i1 = a1.foo1(3); // Synchronous  
21         Future<int> i2 = a1.foo1(4); // Asynchronous  
22         Ptr< Proxy_MyAClass > a2p = new Proxy_MyAClass(2); // ...  
23         Ptr< Proxy_MyAClass > a3p = a1.foo2(*a2p);  
24         func(a1);  
25     }  

\textbf{Listing 2.2:} Part of code generated from the code in Listing 2.1  

The two level reference counting makes proxies very light-weight (in essence they are just smart pointers to the proxy implementation data structure). Copying and destroying proxies locally only requires updating the local reference count. Only when a proxy is sent from one active object to another, the reference count in the actual active object has to be updated. This helps to minimise message passing between active objects. The reference counters are also used to decide when an active object is no longer needed and can be safely destroyed.

It would be impossible to update the reference counts using normal C++ pointers and references. KC++ implements its own proxy pointer and proxy reference template classes. These classes behave exactly as their built-in counterparts, but take care of updating the reference counts. The compiler replaces all pointers and references to active objects with these templates, as can be seen from the code in Listing 2.2.
2.2 Futures

Every active object method call is implicitly asynchronous in KC++. Methods returning `void` do not require any special treatment as there is no return value to return to the caller. For any other return value type, future types are used to allow the actual return value to be returned later when the method execution completes. Futures have also other uses (see Section 2.3). Chapter 5 discusses futures in detail.

The KC++ futures are very similar to futures in Multilisp [Halstead, 1985] and ABC++, and wait-by-necessities in C++. They also resemble IOUs in the Threads.h++ library [Thompson, 1998]. A future type represents another type and is created from a template class parametrised with this type (the “underlying” type). Futures can be “bound” to values which will become available later in the computation. When the value of a future is requested, the future pauses the execution of the requesting thread until the value becomes known, thus forcing synchronisation. A future which doesn’t yet know its value is called a pending future.

2.2.1 Normal futures

The KC++ futures can be assigned to each other and copied (and thus passed as parameters) without having to wait for the value. Futures can also be passed as parameters to other active objects (see Sect. 2.3). This is also possible without synchronisation of pending futures.
CHAPTER 2. OVERVIEW OF KC++

Every future has an implicit type conversion to its underlying type. This conversion function first waits for the value to become available and then returns the value. Futures also have a type conversion in the opposite direction to create a future from the value it represents. This conversion creates a future whose value is already known. This way futures can be used to replace normal variables without having to make any other changes to the code.

In addition to type conversion operators, futures have a member function `value()` to get future's value explicitly (of course an explicit type conversion can also be used). They also contain a member function `wait()` which waits for the value to become available but doesn't return it. Similarly method `isready()` can be used to ask a future whether a value is already available.

When the KC++ compiler generates proxy classes from active classes, it replaces all method return types except `void` with futures of the same type, as can be seen from the code in Listing 2.2 on page 10. However, if a return type is already a future, it is not changed in any way (this enables explicit use of futures in the active classes). With these changes every method call through a proxy can be made asynchronous. When a proxy method is called, the proxy sends a method request to the active object, creates a future which is bound to the eventual return value of the method, and then returns the future without waiting for the method to be completed. From the user's point of view, whether the method call appears to be synchronous or asynchronous depends on how the return value is handled.

To have a synchronous method call, the user can assign or copy the return value of a method to a non-future variable. This causes the implicit type conversion to be used to get the actual return value from the future, and this blocks the process until the value becomes known. This happens on line 20 in Listing 2.1 on page 8.

If the return value of a method is assigned or copied to another future, no blocking is needed and the method call is asynchronous. This is shown on line 21 in Listing 2.1. By using futures everywhere where the actual value of the return value is not needed, the user can delay synchronisation until it is absolutely necessary.

### 2.2.2 void-futures

There is a special type of future called a `void-future` (Future<`void`>) where the type the future represents is `void`. These can be used to achieve pure synchronisation without passing any values between the processes. The void-futures do not have an implicit type conversion or the `value()` method, but they do have `wait()` and `isready()` methods for synchronisation. Section 5.4 in Chapter 5 covers void-futures in detail.

The KC++ does not use void-futures internally, but the programmer can use them to explicitly synchronise with member functions which would normally return nothing. They can also be stored or passed from an active object to another as synchronisation tokens.
CHAPTER 2. OVERVIEW OF KC++

2.2.3 Future sources

In addition to normal futures, KC++ has types called future sources. Future sources are also implemented as a template class, and they act as an explicit way to create normal futures (the implicit way being the return value of an active object method call). In a sense, normal futures are the “destination” of a value, and future sources are the “source.”

When a future source is created, it represents a place-holder into which a value can later be assigned. Normal futures can be created from a future source with a type conversion. Initially, every normal future created from a future source is a pending future without a value. Later, when a value is assigned to the future source using its bind() method, all the futures created from the future source (and all the futures created from these futures) receive the given value.

Future sources are useful when an active object wants to return a value which is not yet known when the method ends. They can also be used for explicit synchronisation. Future sources of type void are especially suitable for synchronisation as they don’t carry any actual value, but act as pure synchronisation objects. Section 5.5 in Chapter 5 goes deeper into future sources.

2.3 Passing parameters to active objects

KC++ active objects do not have to exist in the same address space. This means that the normal C++ parameter passing mechanism cannot be used to pass method parameters to active objects, because normal C++ parameter passing requires a shared address space.

Active objects in separate address spaces communicate with each other using the KC++ message passing subsystem. All member function parameters have to be marshalled into a data stream which is passed as a part of the method request. A method request message consists of the method identifier (identifying the requested method) and a data stream which contains all the marshalled parameters. When an active object receives the method request, it first identifies the method (using the identifier) and then unmarshals the data stream, creating local copies of the parameters.

The KC++ marshalling/unmarshalling system resembles the C++/ marshalling system or the CORBA GIOP protocol [OMG, 1998a, Ch. 13]. However, KC++ does not store any type information (like C++ meta-classes) during marshalling, so a marshalled parameter list contains just the marshalled parameter values concatenated one after another. Storing type information is not needed in a language like C++ because identifying the method already fixes the number and types of the parameters. Each different version of an overloaded method is considered a different method with a different method ID, so the method ID already contains all the information about parameter types. Parameter passing is the subject of Chapter 4.
2.3.1 Normal parameters

The KC++ library contains functions to marshal and unmarshal all C++ built-in types except for pointers and references. This makes passing parameters of built-in types to active objects identical to passing them as parameters to normal objects.

If the user wants to pass his own data types (data structures, non-active objects etc.) to an active object, he must write appropriate marshalling and unmarshalling functions for them. The KC++ library provides the classes OMsg (output message) and IMsg (input message), which represents data streams to and from which data is to be marshalled to using an iostream-like operator <<. The data is unmarshalled from the stream using either an unmarshalling constructor or an operator >>. These can be overloaded for user-defined data types.

The only exceptions to this are pointers and references which cannot be passed as parameters at all, because they would require shared memory (and even with shared memory would pose mutual exclusion problems). However, pointers and references to active objects and references to futures are allowed. These will be explained below.

2.3.2 Future parameters

KC++ future types are allowed as parameters to active object methods. The only requirement is that the underlying type of the future type must also be valid as a parameter. The semantics of passing a future as a parameter depends on whether the future already contains a value or whether it is still pending.

If the future has already received its value, passing it as a parameter is identical to passing its value. As there is a type conversion from the underlying type to the future type, it is also possible to pass instances of the underlying type as parameters to methods which expect future types. Doing this is identical to passing a future which contains the given value.

If a pending future is passed as a parameter to an active object, a copy of the pending future is created at the active object side and this future is bound to the future on the caller’s side. This way, the value of the future can remain unknown even when it is passed as a parameter. Later, when the value of the future becomes known, the KC++ run-time system takes care of forwarding the value to other active objects which contain copies of the future.

2.3.3 Future reference parameters

In addition to return values, reference parameters are a standard C++ way to pass information back from a method. This is not directly possible with active objects because normal reference (and pointer) parameters cannot be allowed. The same effect can be achieved by having a future reference as an active object member function parameter.
CHAPTER 2. OVERVIEW OF KC++

The semantics of future reference parameters is close to the in-out parameters in some languages. When a future is passed as a reference parameter to an active object, the future is passed normally as a parameter. Then the future on the caller’s side loses its value and becomes pending again. It remains in this state until the execution of the method has completed in the active object. At this point the future receives the final value of the corresponding parameter on the active object side.

2.3.4 Active object parameters

Unlike with normal parameters, there are no restrictions on using active object pointers or references as parameters to active object methods. Active objects already have a possibility to refer to each other through proxies. In a sense all active objects occupy the same “active object address space,” so passing active object pointers and references between active objects poses no problems.

It should be noted that KC++ follows the normal C++ parameter passing semantics when passing active objects. If an active object is passed by value, a copy of the active object is created (using the active object’s copy constructor) and that new object is then passed as the actual parameter. When passing pointers and references to active objects, no copies are made. This allows the programmer to use the same coding style in passing both normal objects and active objects as parameters.

2.4 Locking active objects

Quite often a service provided by an active object cannot be completed within one method call, but requires several calls with some processing outside the active object between the calls.

An example of this could be a read buffer handled by an active object. A user first checks whether the buffer is empty (first method call) and then retrieves a value if it was not (second method call). It is important that another user is not able to “steal” the value (and empty the buffer) between the calls.

A normal solution to these problems are object locks. Normally an active object serves method requests in FIFO order (which is often non-deterministic as the order in which requests arrive may be non-predictable). However, when a user locks an active object, that object serves only method requests from that user (in FIFO order) until the lock is released.

KC++ uses lock objects to provide exception safe object locking. When a user wants to lock an active object, he surrounds the code which requires the lock with braces and thus creates a new local scope. At the beginning of this code block he creates a lock object of type ActiveLock using the active object as a constructor parameter.

The lock object makes sure that only the creator of the lock object can access the active object for the lifetime of the lock object. When the lock object is destroyed at
the end of the scope, the lock is automatically released and others are again allowed to use the active object.
Chapter 3

Active objects and proxies

Active objects and their proxy object implementation form the basis of KC++. This chapter describes in detail the KC++ active object model and compares it to other active object based concurrent object-oriented languages.

KC++ uses a class-based active object model. This means that the programmer cannot mark individual objects active, but he can do so for whole classes. After this all objects instantiated from such a class are automatically and always active objects. Other C++-based languages to use a class-based model are ABC++ [O’Farrell et al., 1996] and C++//, for example. Some concurrent object-oriented systems like UC++ [Winder et al., 1996] use an object-based approach where the programmer can make an object of any class active. In C++// both these approaches are possible.

The main reason behind choosing a class-based model in KC++ is that an active object is conceptually different from a normal one. All active object methods are executed asynchronously, and therefore an active object and an otherwise similar normal object should not belong to the same class.

This difference between an active object and a “passive” object is also dictated by the KC++ language implementation. All active object method calls are asynchronous, and methods not returning void return futures as return values. Returning futures is implicit in KC++ (see Section 3.2), i.e. the programmer does not have to explicitly mark method return types as futures (return types can be explicitly marked as futures, but this has a slightly different semantics, see Section 5.2.3). This means that the actual return types of active object methods are different from otherwise identical non-active object methods, making it impossible to combine both in the same class.

3.1 Making objects active

The programmer declares a class active by deriving the class from a base class called Active. After that every object of that class is automatically an active object with its own thread of execution.
CHAPTER 3. ACTIVE OBJECTS AND PROXIES

An active class does not have to be directly derived from the base class Active. It can also be derived from an existing active class, in which case there is a normal subtyping relationship between the classes.

A third alternative is to create an active class from an existing non-active class by using multiple inheritance and deriving the active class both from the non-active class and the class Active. In this case there is no subtyping relationship between the active class and its non-active base class in current KC++. The reasons for this are discussed in Section 3.3. Figure 3.1 shows the possible inheritance hierarchies.

3.2 Proxy classes

When a class is marked active, the KC++ compiler creates a corresponding proxy class and replaces all instances of the active class with the proxy class. In the resulting program all communication with the active object happens through a proxy object created from the proxy class.

The public interface of the proxy class is almost identical with that of the active class, with the exception of future return types described in Chapter 5. However, the internal implementation of proxy objects is quite different. The proxy object does not

---

**Figure 3.1:** Different ways of making a class active in KC++
CHAPTER 3. ACTIVE OBJECTS AND PROXIES

contain any of the data members of the original class, as those data members reside in the active object itself. This somewhat restricts the structure and use of active classes.

The first restriction is that an active class may not contain any public data members because they are not present in the proxy class. It would be impossible to imitate data members in the proxy object because an object has no control over the outside use of public data members. However, use of public data members is normally discouraged in object-oriented programming, so this restriction is not likely to cause many problems.

Even though public data members are not considered good programming practice, it is quite common for an object to access data members of another object of the same class. Since active objects communicate with each other through proxy objects which lack these data members, this kind of access is also impossible. Section 3.4.2 discusses ways to circumvent the limitation.

3.2.1 Internals of a proxy

The core of the a proxy object’s implementation is a data structure called the “proxy implementation structure.” This structure contains enough data to uniquely identify the active object which the proxy represents. This data is also used to send the method requests and future values to the active object.

The actual data stored in the proxy implementation structure depends on the communications library used to send messages from one active object to another. In a distributed environment it might consist of a TCP/IP address and the message port the active object’s method invoker listens to. In a shared memory environment it would be enough to store a pointer to a active object’s method queue and a mutual exclusion semaphore. The important thing is that the data in the structure contains enough information for contacting the active object and deciding whether two proxy objects refer to the same active object. In this thesis, the active objects are simply identified by a single integer for simplicity.

Active object references and pointers to active objects also use the proxy implementation structure. To make creation of such references and pointers as fast as possible, the proxy implementation structure is not stored as a data member inside the proxy object, but the proxy object contains a pointer to the structure. This allows the proxy, references, and pointers to share a proxy implementation. They also maintain a reference count so that the proxy implementation structure is automatically destroyed when it is no longer needed. Active object references and pointers are described in detail in Section 4.3.

When a new active object is created, initially there is only one proxy object. When references and pointers to this active object are created, additional proxies are made. The reference count in the proxy implementation structure allows local copies of the proxies to share the structure. However, when a reference or a pointer to a proxy is passed to another active object, a copy of the proxy must be created in the address
space of the receiving active object. In the current version of KC++ an active object itself contains a reference count of other active objects to contain proxies to the active object. This reference count is used to determine when the active object can be safely destroyed. The proxies send a message to the active object each time the reference count must be incremented (i.e. when the proxy is about to be copied across active objects). Similarly, when the destructor of a proxy notices that it is the last to use a certain proxy implementation structure, it sends a request to the active object asking it to decrease its reference count.

3.2.2 Lifetime of an active object

When code in the program creates a new proxy object, it means that a new active object must be created in the system. The constructor of the proxy takes care of this. It first signals the creation of a new active object process. Depending on the environment the KC++ is implemented on, this might simply mean the creation of another execution thread. On the other hand, on a distributed environment it might require consulting some central load-balancing system on the location of the new active object. After the active object process has been created, the constructor of the proxy sends it a message which requests it to execute the appropriate constructor.

After the proxy object has been created, the program can use the proxy — i.e. call its methods — just as if the proxy was the active object itself. The code in the methods of the proxy relay method requests to the actual active object. This will be described in Section 3.4. The only difference between the public interface of the proxy class and the public interface of the active class is that proxy methods return futures, which allows asynchronous method execution.

In principle the destructor of the proxy could also trigger the destruction of the active object (which includes running the actual destructor of the active class). However, active objects created dynamically with new can be created in one active object and then destroyed with delete using a different proxy in another active object. This means that the destruction of a proxy cannot automatically destroy the active object. Passing active object references to other active objects and asynchronous method execution create also some problems, which are discussed in Section 4.3.1.

Because of these problems the current version of KC++ only destroys an active object after all proxies referring to it have been destroyed. This includes proxies implicitly created by active object references and pointers. This behaviour solves the early destruction problems, but it differs slightly from the normal C++ behaviour and creates a possibility for resource leaks. For these reasons the future versions of KC++ may use a different mechanism for deciding when to destroy active objects (see Chapter 8).
3.2.3 Proxies and inheritance

When active classes are derived from each other, the proxy classes created by the KC++ compiler form an identical inheritance hierarchy. This kind of duplicated inheritance hierarchy is known as the “Rungs of a Dual Hierarchy” pattern [Martin, 1997] and can be seen in Figure 3.2. The proxy hierarchy becomes important when the KC++ compiler replaces all active classes with proxy classes. If the code contains places where a base class reference or pointer is tied to a derived class active object, this code continues to work in the produced code where a proxy base class reference or pointer is tied to a derived class proxy object.

The KC++ class Proxy forms the top of the proxy hierarchy. This class contains everything a proxy object needs to communicate with its active object, i.e. the pointer to the proxy implementation structure and member functions to create and send messages to the active object. Because all these functions are implemented in a common base class, the program contains only a single copy of their code, no matter how many proxy classes are generated. This helps to keep the size of the executables as small as possible.

The actual proxy classes only add necessary member functions to the public interface so that each proxy object can imitate the appropriate active object. The implementation in each of these methods simply creates an appropriate method message, marshals the parameters into the message and then sends the message to the active object. Most of this work is done by calling methods of the base class Proxy, so the size of these methods is quite small. No additional data members are needed in the proxy classes, as the proxy implementation structure in the base class is enough to access all active objects regardless of their class.

Because of the inheritance hierarchy each method has to be implemented only once, in the first base class where it appears. From there it is automatically inherited.
to all derived classes. Dynamic binding is implemented on the active object side, so it is not needed in proxies objects. This is essential because active object pointers and references can cause situations where a base class proxy (for example Proxy_B in Figure 3.2) is used to access a derived class active object (for example D). When a method of the base class proxy object is called, the base class method is always executed because it is the only one available. However, when the message sent by this proxy method is received in the actual active object, dynamic binding is used to find the appropriate method in the active object. This way base class proxies can refer to derived class active objects without problems. This situation is discussed in detail in Sections 3.5.3 and 4.3.1.

3.3 Inheritance and active classes

The class hierarchy of active classes behaves exactly as normal C++ class hierarchies, so pointers and references to active base classes can point to objects of derived active classes. Similarly dynamic binding works as expected with active objects. The only restriction with the current implementation of KC++ is that support for RTTI (Run-Time Type Identification) is not yet implemented.

In KC++ an active class may also have non-active base classes through multiple inheritance. From the viewpoint of the active class's own code this inheritance works as intended, i.e. a method of the active class can call public and protected methods of a non-active base class. This mechanism allows "reuse" of existing normal C++ classes. Appendix A shows an example of this by defining an active buffer class based on an existing buffer.

Using active classes based on non-active base classes is a little bit more restrictive in the KC++. The public methods of the non-active base class are accessible through the active class. Futures are used normally in the return types of these methods to provide asynchronous method calls. However, non-active base class pointers (or references) cannot point to a derived active object. So, outside the implementation of the methods of the active class, there is no subtype relationship between the class and its non-active base class.

The reason for this restriction is the way KC++ active objects are implemented by replacing the actual active objects with light-weight proxies and instantiating the actual active object elsewhere. If the proxy classes would be derived from non-active base classes, they would also inherit the implementation (i.e. data members) of the base class. This would be redundant (as the proxy just acts as a relay between the user and the actual active object), and it would also make the proxy possibly very heavy-weight.

Non-virtual methods of the base class would also be problematic. There is no perfect way to override the implementation of these methods in a derived proxy class, as dynamic binding is only used with virtual methods. However, the proxy class
CHAPTER 3. ACTIVE OBJECTS AND PROXIES

needs to override all methods of the base class because it has to forward all methods calls to the active object.

Another reason not to derive proxy classes from non-active base classes is the fact that every method call to an active object is asynchronous. This would be impossible to achieve for base class methods which return something else than \texttt{void} (or an explicit future), because the base class call would be synchronous (a normal C++ method call) but the derived proxy class call should be asynchronous. This would also cause problems in the type system. Because every non-\texttt{void} (and non-future) return value is replaced with an appropriate future in the proxy class, the interfaces of the proxy class and the non-active base class would no longer be compatible in the C++ inheritance sense (although to the user they are practically identical as the future return value emulates a normal return value).

3.4 Calling active object methods

Figure 3.3 shows how an active object method call is performed using a proxy object. The method call proceeds through the proxy object in the following way (return values are covered in Section 3.6, here \texttt{void} return type is assumed for simplicity):

1. When program code calls a method of the active object, the KC++ compiler transforms this to a method call of an appropriate local proxy object.
2. The method in the proxy creates a message specifying the caller and the method. Then all parameter values are marshalled to the message as well (marshalling is covered in Section 4.1). The message is then sent to the active object.

3. In the active object the message enters a message queue where it waits for its turn. When the message is at the head of the queue (or all messages before it are locked out, see Section 5.6 in Chapter 5), the method invoker reads the message and unmarshals all parameter values into local variables (unmarshalling is also covered in Section 4.1). After this the method invoker calls the method with the unmarshalled parameters using a normal C++ member function call.

When the active object method call completes, and if the method had a non-void return type or the parameters contain future references, the method invoker may send a message containing values to be passed back to the caller. This message is part of the KՀ future binding mechanism and is covered later in Chapter 5. Finally all local parameter values are destroyed and the method execution is complete.

3.4.1 Local method calls

The active object method invocation mechanism described in the previous section is used only when methods of an active object are called from outside. When an active object method calls another method of the same object, the situation is somewhat different. Below these method calls from another method of the same active object are called “local method calls.”

All non-local method calls enter the method queue and are executed in strict FIFO order (with the exception of object locking). This combined with the one-method-run-to-completion semantics of KՀ would cause problems with local method calls. Most local method calls are inherently synchronous in nature, i.e. even if the method call would be asynchronous, the calling method would still have to wait for the return value.

If local method calls would use the normal method queue, this would lead to a deadlock as the calling method would still be executing and waiting for the local method to complete, but the local method would only be taken from the queue and executed after the first method had been completed.

The KՀ system solves this problem by treating local method calls differently from non-local calls. All local method calls are considered privileged and are executed immediately without waiting for the calling method to complete, thus locally ignoring the run-to-completion semantics. This follows the view that most local method calls are calls to private auxiliary functions whose execution should be considered to be part of the execution of the calling method. It is somewhat analogous to Eiffel, where class invariants are not required to hold (or at least they are not checked) for local method calls [Meyer, 1992].
One technical difference between a local and a non-local method call is that the proxy invocation method is not needed in the local case because the active object and the calling code are known to be in the same address space, and so the active object is directly accessible to the caller. K\textsc{C++} utilises this situation by leaving all local method calls untouched, as normal \textsc{C++} member function calls. This removes the overhead of normal active object method invocation as the intermediate proxy call and the method invoker are not needed. At the same this automatically makes local calls privileged as they never even enter the method queue. The solution also allows the actual \textsc{C++} compiler to inline and otherwise optimise local calls normally, which can further speed up the execution of local methods.

If the programmer wants a local method call to be executed as a non-privileged method call, the method can be called indirectly using \texttt{this->method()}. These calls are converted to normal proxy method calls which enter the method queue and are invoked in normal FIFO order. It should be noted that the calling method must not start waiting for completion of the called method, because the called method starts executing only after the calling method has been completed.

### 3.4.2 Accessing private members of other objects

Another implication of leaving local calls as normal \textsc{C++} calls is that the proxy classes do not have to contain private member functions of the active object, as these are not accessed through the proxy. This reduces the size of code generated by the \textsc{K\textsc{C++}} compiler as proxies become smaller.

However, \textsc{C++} allows objects of a class to access private members of other objects of the same class. The most common use of this is to access data members of other objects of the same class. With active objects this is impossible, however, because the data members of another active object are possibly in a different address space. For this reason \textsc{K\textsc{C++}} does not allow any kind of access to the private members of other active objects.

Sometimes it would be convenient to have a special privileged interface between objects of the same class. In \textsc{C++} programmers usually use the private access described above and write special private member functions which are accessed from other objects of the same class. This kind of use of private members is convenient but unfortunately the \textsc{K\textsc{C++}} compiler cannot find out which private member functions are used only locally and which are accessed from other objects. This happens because the compilation is performed separately for each translation unit, so there is no “global picture” of where member functions are called from.

Current \textsc{K\textsc{C++}} offers an inelegant solution to this problem by allowing active objects of one class to call \texttt{protected} member functions of other active objects of the same class. This solution is a compromise as it conflicts with the usual meaning of the keyword \texttt{protected}. However, some distinguishing feature was needed between...
local-only and non-local private methods, so the unusual use of keyword protected was considered an acceptable sacrifice.

3.5 Choosing the correct method

Method messages sent from a proxy object to its active object must identify the method to be called and all the parameters (the message must also identify the return value future, but this is explained later in Section 5.2). The message must contain enough information for the active object to uniquely identify the correct member function and retrieve the parameter values from the message.

Many concurrent object-oriented systems implement this using some sort of restricted metaobject protocol [Kiczales et al., 1991]. In these protocols the method message represents a “method invocation metaobject” — an object which contains data about the name of the method and the values and types of all its parameters. When the method message is received in the active object method invoker, the parameter type information in the message is used to retrieve the parameter values, and the method name is used to find the correct member function. After this the member function can be called normally. It should be noted that metaobject protocols have many other advantages in addition to method invocation. Especially they make customizing and extending the system easier.

The metaobject protocol approach to method invocation is elegant, but increases the overhead needed in parsing the method message. However, a metaobject protocol is not necessarily needed for method invocation in a language like C++, which is fairly strongly typed and uses pass-by-value as its parameter passing mechanism. An alternative is to use a method ID generation scheme which is similar to the name mangling scheme used in most C++ compilers [Lippman, 1996, p. 117].

In this approach the ID of every method is uniquely generated from the method name and the types of the formal parameters (the parameter types are needed because of function name overloading in C++). The method name and formal parameter types do not change, so all method IDs can be generated at compile-time. The method message can then simply consist of the method ID and a bit-stream containing the values of all parameters in the method invocation. No parameter type information is necessary as it is compiled into the method ID. Figure 3.4 on the next page shows how a method message can be constructed from the member function declaration and the actual method call.

In many object-oriented languages the scheme described above would break when polymorphic objects are passed as method parameters. In these cases it is possible that the actual type of the parameter differs from the formal type of the parameter. For example, an object of type D derived from class B could be passed as a parameter to a method expecting an object of type B. If parameters were by default passed by reference (as is the case in many object-oriented languages, but not in C++), the
method ID would identify the parameter as type B even though the marshalled object in the method message would be of type D. This would probably break the message parsing mechanism as the bit-stream containing the parameter would be interpreted incorrectly.

This problem does not apply to KC++ which uses the same default pass-by-value parameter passing mechanism as C++. The reason for this is best described by dividing the possible formal parameter types to categories and discussing them one at a time.

### 3.5.1 Basic types

Basic C++ types like `int, char, double` etc. pose no problems as these types are not polymorphic, and therefore the actual parameter type is always the same as the formal parameter type. The implicit type conversions are not a problem either because they are carried out before the method call.
CHAPTER 3. ACTIVE OBJECTS AND PROXIES

3.5.2 Class types and structs

Class types combined with subtyping and inheritance are usually the reason why a metaobject protocol is needed. In languages based on C++ this also applies to struct types because they are semantically similar to classes.

When an object is passed as a parameter, it is possible that the type of the actual parameter is in fact derived from the type of the formal parameter. In this case the pass-by-value semantics of C++ eliminate the problem. Even though the type of the parameter differs from the type of the formal parameter, the actual value passed to the method is a sliced\(^1\) copy of the actual parameter. This happens because pass-by-value implies creating a copy of the parameter, and in C++ the type of the copy is determined statically, ignoring the dynamic type of the parameter.

Slicing is a common source of problems in C++, but in this case it also means that polymorphism is not an issue when objects and structs are passed as value parameters.

3.5.3 Pointer and reference types

Pointers and references are also capable of expressing polymorphic behaviour and are therefore capable of creating problems with parameter passing. In KC++ pointers and references can be divided into two categories, which are discussed below in their own subsections.

Pointers and references to non-active types

Pointers to basic types, non-active classes or structs cannot be passed as parameters to active object methods, because the objects they point to are not necessarily in an address space which is accessible from the active object. For this reason these pointers and reference do not cause any problems.

Pointers and references to active objects

Pointers and references to active objects are allowed as active object method parameters in KC++, because active objects can be thought to occupy a “global” address space consisting of all active objects in the program. When an active object pointer or reference is passed as a parameter, only the object ID is stored in the message. The format of the ID is the same regardless of the type of the active object. When the method message is received, the method invoker creates a local, possibly sliced proxy

---

\(^1\) Slicing is a term used to denote the operation of creating a base class copy of an object of a derived class. This is possible because the object of the derived class is also an object of the base class and thus is acceptable as a parameter for the base class copy constructor. The implications of slicing are discussed in [Budd, 1997, p. 221], for example.
object in the address space of the receiving active object. Slicing of the proxy object
does not cause problems, however, because dynamic binding is implemented on the
server side, so a correct method is called even if the method is called through a sliced
proxy object.

Figure 3.5 shows a method call where a pointer to an active object of a derived
class B is passed as a parameter to a method expecting a pointer to base class Base.
The receiving active object creates a proxy which is of the generalised type, but that
proxy can still be used to call all base class methods, which are the only methods
visible through a base class pointer.

```
1. a.Method1(b);
   Proxy object A ID=1
   Method1
   Method2
   Method3

2. a
   Proxy object B ID=2
   Method message
   Method1 Base*

3. p
```

**Figure 3.5:** Passing a polymorphic object
3.6 Returning values from active objects

All active object method calls are asynchronous in KC++. This is easy for methods returning `void`, as there is no need to synchronise with the method. Asynchronous methods with a return value are more complicated, because the return value is not available immediately after the method has been called. Many concurrent object-oriented systems like CORBA [OMG, 1998a] and UC++ circumvent this problem by only allowing methods with no return value to be called asynchronously (the new CORBA messaging specification [OMG, 1998b] provides also an asynchronous method invocation (AMI)). An overview of AMI can be found in [Schmidt and Vinoski, 1998]).

As mentioned in Chapter 2, KC++ active object methods use future types to return values from methods. Futures are covered in detail in Chapter 5, but this section discusses briefly the way return values are returned from a method.

For methods returning a value, asynchronous method calls cannot be implemented without some modifications. If an asynchronous method returning an `int` would simply return an `int`, the caller would have no way of knowing when the return value becomes available. An additional synchronisation layer is needed so that the caller can access the return value only after the method has completed.

Futures are such a layer, introduced by R. Halstead in Multilisp [Halstead, 1985]. Futures are place-holders for values which will become available later. They are used for asynchronous return values in many concurrent programming languages and libraries, for example in ABC++ [O’Farrell et al., 1996] and C++. The KC++ futures have also other uses besides return values from asynchronous methods. These uses will be covered later in this section.

The KC++ futures are implemented as a template class. The type the future is used to represent is given to the future as a template parameter. If the original return value from a method is `int`, the corresponding proxy method (generated by the KC++ compiler) has return type `Future<int>`.

Figure 3.3 on page 23 showed how an active object method call works, but omitted the return value passing mechanism. That is shown in Figure 3.6 on the following page. The return value passing proceeds as follows:

1. The caller creates a future variable for the return value. Initially the future behaves exactly as a normal variable (in this case `int`) and is initialised to the appropriate default value (0 with `ints`).

2. When the caller invokes a method in the local proxy object, the proxy code creates a future object which does not contain a value (these kinds of futures are called pending futures in KC++ and will be explained in Chapter 5). The proxy then includes the ID of the future in the method message which is sent to the active object.

---

2However, often synchronisation is wanted even when the method returns no actual value. In KC++ this is also possible with void-futures, see Section 5.4.
CHAPTER 3. ACTIVE OBJECTS AND PROXIES

Figure 3.6: Returning a value from an active object method

3. The proxy returns a copy of the future created during the previous step, and the caller assigns this future to a future variable. The lazy-evaluation semantics of futures allow this to happen even if the future is still pending. The result of this is that the future variable \( i \) becomes pending and is bound to the eventual return value of the method. All the intermediate future objects are destroyed automatically by C++. After this step all attempts to read the value of \( i \) are blocked until its value becomes available.

4. The method is executed in the active object as a normal C++ member function call. At the end of the method the return value is returned to the method invoker using normal C++ return value passing. The return value from the active object may also be a future, i.e. it itself may represent a value that is not yet known.

5. The method invoker creates a “reply message” which identifies the future on the caller’s side and contains the marshalled value for the future.

6. The KC++ library receives the message on the caller’s side and uses it to give the future its value (the part of the KC++ library responsible for receiving future value messages is described in Section 6.2.3). If the caller had tried to access the future’s value and had been blocked, the block is removed the caller can continue executing its code.

The message containing the return value of a method is labelled as a “reply message” above. In reality a special “reply message” type is not needed because the KC++ future library can be used to send the return values between active objects.
Chapter 4

Passing parameters to active objects

By default C++ parameter passing uses pass-by-value semantics to create copies of the actual parameters inside the function. Pass-by-reference semantics is achieved by explicitly passing a reference or a pointer. Pass-by-reference is normally implemented by passing the memory address of the parameter object, which means that it is only meaningful if both the calling code and the function are executed in the same address space. Pass-by-value also requires a shared address space, because the act of creating a copy of the parameter requires access to both the address space of the actual parameter and the address space of the copy.

In KC++ active objects may occupy different address spaces, so parameter passing with active objects methods cannot be implemented exactly the same way as normal C++ parameter passing. However, the KC++ mechanism tries to imitate the C++ mechanism as closely as possible.

4.1 Marshalling and unmarshalling

The KC++ parameter passing is based on marshalling and unmarshalling parameter values to and from method messages. The proxy object on the calling side stores (marshals) all parameter values into the method message. In the receiving end the method invoker reads (unmarshals) the values and creates temporary variables into which the values are stored for the duration of the method call. It should be noted that if the caller and the active object reside in the same address space, marshalling and unmarshalling could be replaced with normal C++ object copying.

Marshalling happens using the C++ operator << and the KC++ class OMsg which represents an outgoing method message. The marshalling syntax resembles the C++ ofstream output syntax. An object o can be marshalled into a message m with the expression

\[ m << o; \]
Marshalling operators can be chained together just like their \texttt{ostream} counterparts.

Unmarshalling is handled a little bit differently in KC++. Like the class \texttt{OMsg} for marshalling, KC++ has a class \texttt{IMsg} which represents a received method message. Unmarshalling an object from this message includes also creation of the object to be unmarshalled. This is achieved by implementing unmarshalling as a constructor which takes an \texttt{IMsg} stream as a parameter. An object \( o \) of class \( X \) can be unmarshalled from a message \( m \) with the expression

\begin{verbatim}
X o(m);
\end{verbatim}

Unmarshalling of basic types must be handled a little bit differently as they cannot have user-defined constructors. For these types KC++ has the unmarshalling operator \( \gg \) with which a value can be read into an already existing basic type with the expression

\begin{verbatim}
m \gg o;
\end{verbatim}

It is also possible to chain these unmarshalling operators together. The KC++ library includes predefined marshalling and unmarshalling operators for all basic types which are acceptable as active object method parameters.

\section{Different types of parameters}

The different ways to pass parameters to active object methods can be divided to the following six categories.

\subsection{Basic types}

Basic types such as \texttt{int}, \texttt{float}, \texttt{bool}, etc. are passed as parameters just as with normal C++. Their values are automatically marshalled into the method message by the proxy object and unmarshalled at the other end. This mechanism is indistinguishable from normal C++ pass-by-value (except that it is slower).

\subsection{Non-active user-defined classes and structs}

User-defined class and struct types require little extra work from the programmer as KC++ does not have built-in mechanisms for user types. In order to be able to pass an object of a user type to active object methods, the programmer has to provide his own marshalling and unmarshalling mechanisms for the type.

For marshalling to work for type \( X \), a marshalling operator must be defined. The signature of this operator is

\begin{verbatim}
OMsg& operator << (OMsg& omsg, const X& x);
\end{verbatim}
CHAPTER 4. PASSING PARAMETERS TO ACTIVE OBJECTS

The operator should return its first parameter as its return value so that chaining is possible. For most struct and class types writing the operator is easy, as marshalling can be implemented by marshalling all data members into the OMsg stream.

Unmarshalling requires a constructor which receives the IMsg stream to unmarshal from as a parameter. The signature of this constructor for class \( X \) is

```cpp
class X
{
    public:
        X(IMsg& imsg);
};
```

The constructor is usually easy to implement as it can pass the IMsg stream to the constructors of those data members that are of struct or class types. Data members of basic types can be unmarshalled in the constructor body using the `>>` operator. User-defined classes and structs can define the unmarshalling operator `>>` in addition to the unmarshalling constructor, but the KC++ itself never uses this operator. However, user code may call this operator when implementing unmarshalling mechanisms for other classes and structs.

Listing 4.1 on the next page shows a simple class and its marshalling operator and unmarshalling constructor.

### 4.2.3 Enumeration types and non-user-defined types

Enumeration types, classes, and structs not defined by user — from third party libraries or the standard C++ library, for example — have to be handled a little bit differently. Marshalling can be implemented normally with the `<<` operator, but unmarshalling requires a different approach because it is often not possible to add unmarshalling constructors to third party code.

Unmarshalling for these kinds of types is handled by writing unmarshalling operators (```) for these types and declaring them as special cases with KC++ library macro SPECIAL_UNMARSHAL. For type \( X \), the syntax of these is

```cpp
IMsg& operator >> (IMsg& imsg, X& x);
SPECIAL_UNMARSHAL(X)
```

The SPECIAL_UNMARSHAL declaration must appear in the same header file where the `>>` operator is declared. In addition KC++ requires that these types have a default constructor which can be called to create a “virgin” object into which the value can be unmarshalled with the `>>` operator. All enumeration types have this automatically, so this is an issue only with third party types. The need for SPECIAL_UNMARSHAL is further discussed in Section 6.2.5.

Listing 4.2 on page 36 shows an enumeration type and its marshalling and unmarshalling operators.
4.2.4 Active classes

Passing active objects as parameters requires nothing from the programmer. The semantics of passing an active object is similar to passing a non-active object to a C++ function, i.e. the pass-by-value semantics causes a copy of the parameter to be created. The proxy object is responsible for creating the new copy of the active object. Section 4.4 describes passing active objects by value in more detail.

4.2.5 Pointers and references to active objects

It is normal C++ practice to pass object pointers and references as parameters instead of actual objects. In fact, pass-by-reference semantics is more natural to object oriented programming, so object reference and pointer parameters are a rule rather than an exception. For this reason it was considered important that pointers and references to active objects can be passed to other active objects.

From the programmers point of view passing active object pointers and references
CHAPTER 4. PASSING PARAMETERS TO ACTIVE OBJECTS

Listing 4.2: Marshalling/unmarshalling of an enumeration

```cpp
enum Colour {RED, BLUE, GREEN};

OMsg& operator << (OMsg& omsg, const Colour& c)
{
    omsg << static_cast<int>(c); // Store colour as an int
    return omsg;
}

IMsg& operator >> (IMsg& imsg, Colour& c)
{
    int temp;
    imsg >> temp; // Read the value into an int
    c = static_cast<Colour>(temp); // Convert back to a colour
    return imsg;
}
SPECIAL_UNMARSHAL(Colour)
```

requires no extra work. The way this is achieved internally in KC++ is described in Section 4.3. It should be noted that passing active object pointers and references create no synchronisation problems. Each active object automatically has one-method-run-to-completion behaviour, so an active object may be used by several other active objects at the same time.

4.2.6 Pointers and references to non-active types

Pointers and references to non-active types can only function within a certain address space. Because an active object may occupy a different address space than the caller, passing pointers and references to non-active objects and data is not allowed in KC++. However, it is possible to pass a reference to a future, see Section 5.3.

Another reason not to allow pointers and references to non-active types is the fact that all active object method calls are asynchronous. Even if the separate address space issue could be solved, there would still be a problem when several active objects would be able to simultaneously access and modify the object at the end of the pointer. This is clearly unacceptable as non-active types have no built-in mutual exclusion mechanisms like active objects.

4.2.7 Futures and references to futures

As active objects, futures can be thought to occupy a “global address space” and thus they can be passed between active objects. Future types and references to futures can
be used as active object method parameters if their underlying type is also acceptable as an active object parameter. This requirement comes from the obvious fact that the underlying object must be marshalled and unmarshalled as part of the future marshalling and unmarshalling. Passing futures or future references does not otherwise require anything extra from the programmer.

The exact implementation and semantics of future parameters are explained in Section 5.3.

4.3 Pointers and references to active objects

Pointers and references to objects are very common in C++. In fact, in many programs objects are accessed more often through pointers or references than by their declared name. For this reason it is important that pointers and references to active objects behave just as pointers and references to normal objects, or at least imitate the behaviour of normal pointers and references as closely as possible.

All access to active objects happens through proxy objects in the client’s own address space, so at first it may look like normal pointers and references to proxy objects are enough, and no extra mechanisms are needed for pointers and references to active objects.

The problem lies in the fact that pointers and references to objects are very common as parameters. Especially it is very common in object-oriented programming to pass a pointer or reference to one object as a parameter to a method of another object. In KCC++ this means that pointers and references to active objects must be allowed as parameters to an active object method. However, all proxy objects are local to each active object, so when a pointer or reference is passed between active objects, the proxy at the end of the pointer or reference must be duplicated in the receiver.

For example, when a pointer to active object A is passed as a parameter to active object B, the KCC++ system must create a copy of the A-proxy in B’s address space and then use a pointer to that proxy in B’s code. If KCC++ would use normal C++ pointers to proxies, this would lead to several problems and inconsistencies:

- Two pointers should compare equal if and only if they point to the same object. This means that all pointers to the same active object in each address space should point to the same proxy object. So each time an active object pointer is passed between active objects, the receiver would have to check whether there already exists a proxy object for that particular active object and use that proxy if possible. Otherwise two pointers to the same active object could point to different proxies and so not compare equal. Such “proxy search” would make passing active object pointers very slow when there are many active objects.

- The fact that there is one active object but several copies of the proxy makes
destroying the proxies complicated, because the lifetime of the proxy is not necessarily the same as the lifetime of the active object.

The local copies of the proxies have to be created dynamically, because the \textit{KC++} system cannot know when the copy is no longer needed. The lifetime of the original pointer or reference parameter is known, but of course the program may create additional pointers and references to the proxy and the lifetime of these is unknown. This means that in order to destroy the proxy object, the system must know when there are no more pointers or references to the proxy. This cannot be achieved with \textit{C++} pointers and references.

The \textit{KC++} system solves these problems by providing its own pointers and references as template classes, and by replacing all pointers and references to active objects with objects of these classes. The \textit{KC++} pointer and reference classes take care of properly destroying the proxy at the end of the pointer or reference when the lifetime of the pointer or reference ends. Similarly comparing two \textit{KC++} pointer objects returns “true” if and only if the proxies at the end of the pointers refer to the same active object. The following sections explain the features of \textit{KC++} pointers and references.

### 4.3.1 References to active objects

In \textit{C++} objects and references to them behave almost identically. Both have the same interface and all actions on a reference are directed to the object which the reference refers to. In \textit{KC++} active objects are replaced with proxy objects which are practically just another kind of reference objects. This makes references to proxies and proxies themselves resemble each other even more.

There are some differences between the behaviour of references and actual objects. The most important ones are listed below. They are quite self-evident, but they have to be taken into account when \textit{KC++} imitates references with its own class template:

- References are usually very light-weight compared to objects. The memory consumption of a reference is normally only a few bytes, whereas an object may need any amount of memory depending on the size and number of its data members. This means that creating and copying a reference is much faster than doing the same operations on the object.

- When a reference is created from an object (or another reference), it simply binds itself to that object (or the object the reference refers to). On the other hand, when an \textit{object} is created from another object, it is copy-constructed, resulting in two objects.

- When a reference is destroyed, it simply disappears without affecting the object
it has been referring to. This is of course different from destroying an object itself.

- A reference of type \( X \& \) can refer to an object of type \( X \) or any object derived from class \( X \). The static type of the reference does not tell the type of the object it refers to.

When a KC++ program is run through the compiler, all references to active classes are replaced with an instance of a reference template. In other words, each type \( X \& \) where \( X \) is an active class is replaced with type \( \text{Ref}<\text{Proxy}_X> \), where \( \text{Proxy}_X \) is the type of the proxy object.

The KC++ reference template is defined by deriving it from the proxy class it gets as its parameter, i.e. the beginning of the template definition is

\[
\text{template}<\text{typename } T> \\
\text{class Ref : public } T
\]

By deriving each active class reference from the proxy, the reference automatically inherits the public and protected interface of the proxy. That is enough in KC++, because active objects do not have access to each others private members. This derivation is very convenient because it allows the definition of the reference template to be placed in the KC++ library instead of having to be created by the KC++ compiler.

The derivation takes care of having the same interface in an active object reference and a proxy. However, there are still the differences listed earlier between objects and references. These are taken into account in the following manner:

- KC++ proxy objects are already very light-weight, containing only a pointer to the reference-counted proxy implementation. That makes the references derived from the proxies just as light-weight, so in this respect KC++ reference objects resemble C++ references.

- When a reference object is created, its constructor always gets a proxy (or another reference object, which is derived from a proxy) as a parameter. The constructor of the reference calls a special constructor of the proxy base class. This special constructor binds the new proxy object to the same active object as its parameter. This is a very fast operation as it only requires sharing the proxy implementation and incrementing its reference count. This way creation of a reference object achieves the same effect as creating a C++ reference. The special constructor is written to each proxy class by the KC++ compiler.

- When a reference object is destroyed, its base class proxy is also destroyed. It detaches itself from the proxy implementation, and if it was the last to use that implementation, it signals the active object. The active object destroys itself when its own reference counts reaches zero.
This behaviour means that currently proxies and references behave identically when they are destroyed. The active object is not destroyed even if its “primary” proxy (the proxy that created it) is destroyed, if there are still references to it. This behaviour makes “dangling” references impossible, but also makes KC++ programs behave differently from normal C++ programs. This problem is discussed in detail later in this section.

- The data inside each proxy class is identical regardless of the active class the proxy represents. The functional interface of each proxy is different (depending on the interface of the active class), but each proxy just contains a pointer to the proxy implementation structure, which in turn just identifies the active object.

This similarity in proxies makes it possible to create a proxy of a base class type and bind it to an active object of a derived class. The method invoker inside the derived class active object process understands all method messages created by a base class proxy, so the base class proxy can be used to invoke any base class method in the derived class object. Dynamic binding is implemented on the server side, so it also works automatically.

When a KC++ base class reference object is created from a derived class proxy, everything works as it should. The base class proxy part of the reference object shares the proxy implementation with the derived class proxy, but it poses no problems since the implementation is identical for all proxies.

Perhaps the most common use for references is to use them as parameters, allowing a function to use an object created elsewhere. Normally this means that the object is first created, then passed to a function as a reference parameter and then later destroyed. The function can safely use the reference because it knows that there is always an object at the end of the reference.

The situation becomes more complex when concurrency and asynchronous method calls are added to the picture. When the method and the caller of the method execute concurrently, it would become possible to create an object, pass its reference to another active object and then destroy the object before the method has completed it execution. This would mean that references to an object could suddenly become “dangling” references.

Listing 4.3 on the next page shows a simple example where an active object is passed as a reference to a method of another active object. If the code of the function f would be executed as a normal C++ program, the program would work without problems. On the other hand, when it is compiled as a KC++ program, the method call on line 22 becomes asynchronous, allowing both the caller and the method to execute in parallel. Therefore it becomes possible (probable, in fact) that the actual proxy object is destroyed in f before the method has completed and stopped using the active object. This kind of behaviour is not possible in a sequential program, making the problem hard to notice. There are several possible solutions to this problem:
class A : public Active
{
public:
  void action();
};
class B : public Active
{
public:
  void method(A& a);
};
void B::method(A& r)
{
  r.action(); // The object r refers to must exist!
}
void f()
{
  B b;
  A a;
  b.method(a);
  // a is destroyed here
}

LISTING 4.3: Object references and asynchronous calls

1. “Dangling” references could be allowed. In this case the programmer would just be told to avoid such problems by designing programs carefully.

   This solution is very easy technically, but it means that programmers have to make sure that objects passed to active object methods as references are not destroyed before the method is completed. This would require additional manual synchronisation in the program.

2. An active object could “stay alive” until all references to it are destroyed, even if the “primary” proxy object which created the active object is destroyed before the references.

   This alternative makes sure that there are no “dangling” references, but on the other hand it means that active objects are not necessarily destroyed when the programmer expects. To the client code the proxies should be identical to
the actual active objects. This makes letting the active objects exist after their “primary” proxy has been destroyed significantly different from the way things are done in normal C++.

There is another problem in addition to the inconsistency described above. Cyclic references would create situations where two active objects refer to each other and for this reason neither is ever destroyed. This problem is identical to problems in all reference-counting based garbage collection systems.

3. Destruction of the “primary” proxy could be delayed until all references to the active object have been destroyed. This would mean pausing the execution of the thread owning the primary proxy until all references have been destroyed.

This solution would make destroying active objects through proxies behave the same way as destroying normal objects. However, cyclic references would still be a problem and cause deadlocks instead of object leaks.

4. In addition to the solutions listed above, it would also be possible to modify some mark-and-sweep garbage-collection algorithms to find out whether an active object could be destroyed safely. However, distributed garbage-collection systems are usually very heavy-weight. The order of destruction is also important because of destructors, and many garbage-collection systems do not take that into account.

The current version of K C++ currently uses solution number 2 — destroying active objects only after all references to them are destroyed. However, it seems that cyclic references are so common in C++ programming that they cause more problems than “dangling” references. Chapter 8 briefly discusses possible future changes to K C++.

4.3.2 Pointers to active objects

Pointers to active objects are in many ways similar to references. Pointers in C++ are also very light-weight, creating them does not create actual objects, and a pointer to a base class can also point to an object of a derived class. However, pointers have some additional characteristics:

- A pointer to an object can be obtained by creating an object dynamically with new or by applying the “address-of” operator & to an object or a reference.
- A pointer can be empty (or “null”), i.e. not pointing to anything. The user can test this by comparing the pointer to zero (if (p == 0)) or by directly using the pointer as a condition (if (p) or if (!p)).
- Objects created dynamically with new can (and should) be destroyed explicitly through a pointer with delete.
- The object is accessed through the pointer using operators * and ->.
• If a pointer points to an object in an C++ array, the pointer can be moved inside the array using pointer arithmetics (operators ++, −− etc).

The KC++ compiler replaces all pointers to active objects with instances of a pointer template. Each occurrence of type X*, where X is an active class, is replaced with a type Ptr<Proxy_X>, where Proxy_X is the type of the proxy object.

The pointer template is defined as a class which internally maintains its own proxy object which in turn refers to the actual active object. The pointer characteristics are implemented in the following manner:

• A KC++ pointer object can be initialised from a C++ proxy pointer returned by the new operation. In addition to this, all proxy classes implement their own operator & which return a KC++ pointer object instead of a C++ pointer.

• If the pointer object is empty, it simply contains no proxy object. The pointer object implements the “logical not” operator ! and a conversion to a void* pointer. With these the user can test the emptiness of a pointer object in the same way as with normal C++ pointers.

• The C++ delete operator cannot be used, because it requires a C++ pointer as its parameter. For this reason the KC++ compiler replaces all active object pointer delete operations with a call to the pointer object’s method destroy(), which simply destroys the pointer’s proxy object and empties the pointer object.

• The pointer template classes implement their own operators * and −− which return a reference to the pointer’s proxy object. This way an active object can be accessed through a pointer object in the same way as through a normal C++ pointer.

• The current version of KC++ does not support pointers to active object arrays, so pointer arithmetic is not supported. Allowing pointer arithmetic would mean that if a pointer to an array would be passed as a parameter between active objects, the KC++ would have to copy the whole array from one active object to another, as there would be no way to know whether the parameter pointer is later moved inside the array.

When a pointer object is created from an active object proxy, it creates a local duplicate proxy object referring to the same active object. When a pointer object is destroyed, it automatically destroys its local proxy.

Each pointer object creates its own local proxy because of parameter passing. Figure 4.1 on the following page shows a situation where a pointer to an active object is passed as a parameter between two active objects. In this kind of situation the receiving active object does not necessarily have any proxy for the active object the pointer points to. For that reason creating a copy of the pointer object in the receiving end
has to include creation of the proxy also. The proxy must also be destroyed properly when the pointer object is destroyed.

The easiest way to make sure the additional proxies are always created and destroyed properly is to let each pointer maintain its own proxy object, as current KC++ does. This means that creating a pointer means the additional overhead of creating an extra proxy even when the pointer object could point to an already existing proxy. This overhead is very small, however. Each proxy object just contains a pointer to a reference counted proxy implementation, so copying a proxy just means copying the proxy implementation pointer and incrementing the reference count.

### 4.4 Passing active objects by value

When a function in C++ takes an object as a normal parameter (i.e. not a reference or a pointer to an object), pass-by-value semantics is used. This means that a copy of the actual parameter object is created in the function. This copy is initialised using its
CHAPTER 4. PASSING PARAMETERS TO ACTIVE OBJECTS

copy constructor, using the original object as a parameter. This copy of the parameter object is automatically destroyed when the function returns.

KC++ tries to follow normal C++ semantics as closely as possible. This means that passing an active object as a parameter should have a similar effect as passing a normal object. When an active object is passed by value, a copy of the active object is created and constructed using its copy constructor.

When the function to be called is a normal function or a member function of a non-active object, the normal C++ pass-by-value mechanism is sufficient to take care of creating the copy. The KC++ compiler changes the parameter type of the function from the active class to a proxy class. When the function is called, the creation of the “copy” parameter proxy using the copy constructor automatically triggers the creation of another active object.

The situation is a little bit more complex when an active object is passed as a value parameter between active objects. Figure 4.2 on the next page shows the situation in a case where an active object b is passed as a value parameter to method method1() of another active object a.

1. First the proxy of the active object b is passed as a value parameter to the respective method method1() in the proxy of a. At this point the normal C++ pass-by-value creates a copy of the b proxy.

2. The new proxy creates a new active object and invokes its copy constructor. The copy constructor receives a reference to b as a parameter — this uses the active object reference parameter mechanism described in Section 4.3.1.

3. Next this new proxy is marshalled into the method message. In practise this just means storing the ID of the active object and signalling the active object to increase its internal reference count. Then the method message is sent to a. After this the proxy method is completed and the local parameter proxy is automatically destroyed. This doesn't destroy the actual active object, because it's reference count was already increased during marshalling.

4. When the method message is received in a, the method invoker unmarshals the message and creates a local proxy object referring to b. This proxy is then passed as a parameter to the actual active object method. At this stage pass-by-value must not be used, because it would result in the creation of another copy. The KC++ compiler solves the problem by replacing all active object value parameters with reference parameters in the actual active object method implementations.

5. When the method completes, the method invoker destroys the parameter proxy, which signals the copy of b, which in turn destroys itself.

Passing an active object as a value parameter is clearly a very heavy-weight operation as this example shows. Objects are quite seldom passed by value in object-
oriented programming, but if it is needed, value passing has the same semantics in KC++ as in normal C++.

**FIGURE 4.2:** Passing an active object by value
Asynchronous method calls and futures

Asynchronous method calls are a common way of achieving concurrency in concurrent object-oriented systems, including K\textsc{C++}. Proxy objects make asynchronous method calls very easy to implement, as the proxy object can simply send a method request to the active object and then return. However, in many languages (including \textsc{C++}) methods are also allowed to return values, and these return values force the caller and the active object to synchronise with each other at some point, so that the result of the method can be transferred from the active object to the caller.

Some object-oriented methodologies (the one presented in [Meyer, 1997], for example) divide all methods to two categories — \textit{commands} which are allowed to change the object but not return any values and \textit{queries} which are only allowed return data from the object but not modify it. As most computationally intensive functionality often resides in commands, it is very natural to add concurrency only to these methods. This solution has the additional benefit that return values are not a problem as all the value returning methods — queries — are still synchronous.

Some concurrent \textsc{C++}-based systems like U\textsc{C++} use the same solution and restrict asynchronous operation only to methods with \texttt{void} return types. However, the division of methods to commands and queries is not natural in \textsc{C++}, where it is quite common to return values from computationally intensive operations.

In other systems a single method call can be either synchronous or asynchronous. The synchronous and asynchronous method calls have a different call syntax, letting the programmer decide which way to call the method. ABC\textsc{C++} is an example of this kind of a language. There is a slight conceptual problem with this approach: synchronous and asynchronous methods have different completion characteristics, and therefore it is questionable whether a single method should (or can) be both synchronous and asynchronous depending on the call syntax.
An asynchronous method call is clearly the less restrictive mechanism as it does not say when the method actually gets completed. A synchronous call can then be thought as an asynchronous call followed by a wait for completion. This does not make the synchronous call any less efficient as a similar return value message is needed in both cases to get the return value back from the active object. KC++ uses this approach and makes all active object method calls automatically asynchronous. The user can then decide whether to wait for the return value immediately (a synchronous-looking call) or to leave the return value to be received later (a “real” asynchronous call). Both these alternatives have a very natural-looking syntax, so C++ programmers should “feel at home” with both synchronous and asynchronous calls.

The fundamental mechanism behind asynchronous method calls in KC++ are futures. These are “wrappers” first introduced by Halstead in Multilisp [Halstead, 1985]. They are widely used in several concurrent C++ systems, like ABC++ [O’Farrell et al., 1996], sometimes under different names like “wait-by-necessities” in C++/ [Caromel et al., 1996] or “IOUs” in the Threads.h++ library [Thompson, 1998].

From the caller’s viewpoint futures can be seen as a “placeholder” for a value that will be eventually received from an asynchronous method call. From the method’s viewpoint a future is a “promise” to send a value to the caller.

In KC++ futures are a template class which can be used to instantiate “placeholders” for values of almost any type. These values do not have to be available at the time when a future is created, but they may become available later, for example when a method completes its execution. In many other systems futures are only used to receive return values from asynchronous method calls, but the KC++ future types can be used fully as “first-class citizens” — they can be passed as “delayed parameters” to other active objects or to receive additional return values using future reference parameters.

The KC++ futures do not have to get their values from method return values. Programmers can create future sources which act as the origins of the values the futures represent. This way the programmer can use futures for explicit synchronisation and delayed value passing.

5.1 Principles of futures in KC++

The KC++ future type is a simple template type whose only parameter tells the “underlying type” of the future, i.e. what type the future is meant to represent. The name of the template is Future, so a future type for integers is marked Future<int>, and a future type for string objects is Future<string>.

In order to be able to create futures with a certain underlying type, this type has to satisfy certain requirements:
CHAPTER 5. ASYNCHRONOUS METHOD CALLS AND FUTURES

- The type must have a working copy constructor
- The type must be allowed as a parameter between active objects. This means that it must have a marshalling constructor (\(<<\)) and an unmarshalling constructor (\(>>\)) and a default constructor (see Section 4.2.2).
- If any of the comparison operators \(==\), \(!=\), \(<\), \(<=\), or \(>\)\(=\) is defined for the type, then for all values \(t\) of type the expressions \(t==t\), \(t<\text{=}t\) and \(t>\text{=}t\) should always be \text{true} and similarly \(t!\text{=}t\), \(t<t\) and \(t>t\) should always be \text{false}. This requirement is needed so that future comparisons can be optimised in certain situations. However, it is very unlikely that any reasonable type would not satisfy this requirement.

If a type \(T\) meets these requirements, then futures of type \(\text{Future}<T>\) can be created. As active object method calls return futures as their return values, the requirements also apply to any type that acts as a return type of an active object method.

Once a future has been created (either explicitly by the programmer or implicitly by the KC++ system), it acts as a placeholder for a value and can be used to access this value. A future can be in two states, \textit{pending} or \textit{ready}. A pending future has not yet received its value. When the value is received, the future becomes ready.

If a future is still pending (i.e. its value is not yet available) when an attempt is made to access the value, the access methods in the future wait and suspend the execution until the value becomes available. This way the program can continue its operation normally until it really needs the return value of an asynchronous method, at which point the future automatically forces the program to wait for the value.

### 5.1.1 The interface of futures

A future type \(\text{Future}<T>\) has the following interface:

- Constructors: The constructors allow a future to be created with a known value or from another future. They are explained in more detail in Section 5.1.4.
- Assignment: A value of type \(T\) or another future can be assigned to a future. Assignment is discussed in Section 5.1.5.
- Destructor: When a future is destroyed, it automatically takes care of the destruction of it’s value, if any. Destruction of futures is covered in Section 5.1.6.
- \textbf{bool} \texttt{isready()} \textbf{const}. A future can be asked about its readiness with the syntax \(f\text{.isready()}\). The function returns \text{true} if the future is ready and has already received its value, \text{false} if it is still pending. This function should only be used to check the status of a future occasionally. It should never be used in a tight loop to wait for the value (the method \texttt{wait} is for this purpose).
• **void wait() const**: If the status of a future \( f \) is ready, \( f.wait() \) returns immediately without doing anything. If \( f \) is pending, the call halts the execution of the current thread until \( f \) receives the value and becomes ready. Quite often a future can be used without calling `wait` explicitly, as the value accessing methods call it automatically.

• **const T& value() const**: The call \( f.value() \) first calls `wait` to make sure \( f \) is ready, and then returns a constant reference to the future’s value. This function is usually called internally by the type conversion function described below, but it can also be used to explicitly ask for the value of a future.

• **operator const T&() const**: When a type conversion from a future to its underlying type is performed, the conversion function simply calls `value` to wait for the value of the future and then return it.

• **bool isSame(const Future<T>& )**: Two futures can share a value (regardless of whether the value is pending or not). This happens when a future is assigned to another future or a copy of a future is created (see Sections 5.1.4 and 5.1.5). For two futures \( f_1 \) and \( f_2 \), the call \( f_2.isSame(f_1) \) returns `true` if the two futures share their value, and `false` otherwise. This test can be performed on pending futures without waiting for the value.

• **Comparisons**: If the underlying type supports comparisons (==, !=, <, >, <=, or >=), these comparisons can also be used on the future itself. The comparisons invoke the same comparison operation on the values of the futures, waiting for the values to become available, if necessary.

There is one important exception: If two futures share the value (`isSame` would return `true`), then `==, <=, >, >=` immediately return `true` and `!=, <, and >` return `false`. This happens without waiting for the value to become available or invoking the comparison operations on the already available value (which would only compare the value with itself). The advantage of this behaviour is that if two pending futures share the same value, their equality can be resolved without waiting.

### 5.1.2 Using futures

The kC++ future mechanism is designed with two goals in mind: First, it should be relatively easy to add concurrency to an existing program by replacing objects with active objects and necessary normal variables with futures. This should be possible with as few limitations as possible. Secondly, the programmer should be provided with an interface to manually control asynchronous operations, if needed.

The first goal is mostly achieved through the constructors and the assignment, type conversion and comparison operators. These make it possible in many cases to use a future variable almost as a variable of the underlying type. A future variable can
be initialised with a value of the underlying type, or such a value can be assigned to a future variable later. The value of a future can be read from a future automatically through its type conversion operator, and two futures can be compared, if such a comparison exists for the underlying type. This way, if no asynchronous operations are invoked, using future variables in place of normal variables does not in many cases affect the behaviour of the program at all.

If an asynchronous operation results in a pending future and that is used in an expression where a value of the underlying type is needed, the execution of the thread is automatically suspended until the value of the future is received, and then the value is used in place of the future. The implicit conversion makes it often possible to replace variables of type \( T \) in existing code with futures of type `Future<T>`, allowing the program to automatically benefit from asynchronous method calls.

Listing 5.1 on the following page shows a small program where all instances of the type `int` are replaced with futures, without affecting the program in any way (except for a small performance cost). Of course such a change is absurd without adding concurrency, but it demonstrates how close future variables are to normal variables.

If active objects are used to add concurrency to the program, the program code would have to be changed very little. In this example, classes `Generator` and `Functor` can be made active by deriving them from class `Active`. After this modification their method calls are automatically asynchronous and return futures. Other changes are not necessary.

Because futures can be assigned and copied freely without causing synchronisation, the program in Listing 5.1 runs without synchronisation until the future values inside the vector are added up in the C++ library function `accumulate`. There a variable of type `int` is created (the type is dictated by the type of the initial value 0), and each value in the vector is added to the variable. This expression `int + Future<int>` causes the compiler to invoke a type conversion from the future type to integer, which causes synchronisation and halts the execution of the program until each value in the vector becomes available.

Although the automatic synchronisation during type conversion is often enough, sometimes it is desirable to explicitly control synchronisation. This can be done with methods `isReady` and `wait`. If synchronisation with the completion of an asynchronous operation is wanted before the return value is actually needed, calling `wait` on the return value future halts the program until the method has completed. This may sometimes be necessary to coordinate the execution of active objects, or to simply “catch up” with active objects to prevent an excessive amount of pending operations.

The polling function `isReady` is probably less useful than `wait`. With it a program can start an asynchronous operation and periodically check whether the operation has been completed, performing some other tasks while “waiting.”

Explicit synchronisation becomes more important when future sources are used in the program to achieve user-controlled synchronisation and notification. This subject is discussed in detail in Section 5.5.
class Generator
{
public:
    int firstiter();
    int enditer() const;
    int nextiter();
};

class Functor
{
public:
    int transform(int iter) const;
};

typedef Future<int> Fint;

Fint accumulate(Generator& g, Functor& f)
{
    vector<Fint> fvec;
    Fint iter = g.firstiter(); // Conversion to future
    // Generate values to an array
    while (iter != g.enditer())
    {
        Fint value = f.transform(iter); // Conversion both ways
        fvec.push_back(value);
        iter = g.nextiter();
    }
    // Add values using STL's accumulate
    Fint result = accumulate(fvec.begin(), fvec.end(), 0);
    return result;
}

LISTING 5.1: Example where ints have been replaced with futures
5.1.3 Futures as constant references to “system-wide” data

Values of futures are normally created in one active object and later used in another. Another way to look at the same thing is to think of these values as “system-wide data” which is available to every active object in the program. In this way of thinking futures are simply local objects used to access the global data. Of course the concept of system-wide shared data is not directly possible in a system with no shared memory, but the concept is quite appealing in a system like KC++, where active objects already represent “system-wise objects.”

If values of futures are considered system-wide, the way these values can be modified becomes important. If no shared memory is available, the values would either have to reside in a “server object”, which would serve read and write requests from actual future objects, or each future could keep its own copy of the data and all modifications would have to be sent to each future object “sharing” the value. Parametric shared regions in ABC++ [O’Farrell et al., 1996] are a combination of these approaches. There a server keeps a “master copy” of a shared value and takes care of distributing changed values between clients.

Both these approaches have their problems. The server approach would practically make each future value an active object, and this would mean that even reading the value of a future (or asking whether the value exists) would include the overhead of message passing etc. between the future server and the reader. When futures are used in active object return values, this would increase the amount of message passing considerably, making the approach impractical.

The approach where each future object keeps a copy of the value is very efficient when only reading is required. However, when the value is modified through one future object, all other future objects have to be notified of the new value. This causes a lot of message passing, and additionally every future object would have to know all other future objects sharing the value in order to be able to send message to them. This would increase the size of future objects, and each time a new future object is created, every other future would have to be notified of this event, too. All this makes the approach quite impractical, too.

The problem of copying the value becomes much easier if the value cannot be changed after it has become known, i.e. the value can only be initialised but not modified. In this case the creator of the value has to send a copy to pending future objects, but after that there is no need for communication between the futures. Only the creator of the value has to know where to send it, otherwise the futures do not have to be aware of each other. Figure 5.1 on the next page shows together all the three different future implementation possibilities.

KC++ uses the “constant copy” approach to implement futures. Making future values unmodifiable is quite justifiable, because the principal reason for futures is return value passing, and there is no reason to “change the return value” after a method has been completed. In practise, a future object in KC++ can be thought as a “constant
reference” to a system-wide value. The analogy is very concrete because the type conversions in future classes allow futures to be implicitly converted to a constant reference to their value.

Each time a pending future is copied from one active object to another (through parameter passing or as a return value), the KC++ run-time system remembers that when the value of that future becomes available, the value must be forwarded to the other active object. When the value finally is available (when a method returns a value or when a future source is bound to a value), all appropriate local pending futures in the same active object are first given the value. The run-time system then checks to see where these futures have been copied, and sends a message containing the value to other appropriate active objects. After this the run-time system can discard the forwarding information. The run-time system in other active objects does the
same, and so the value of the future propagates from one object to another, following the same path as the original pending future.

There are some differences between constant references and futures, however. First, a new value (pending or not) can be assigned to a future. This causes the future to "bind" itself to the new value and abandon the old one. This is not possible with references, which always refer to the same value for their entire lifetime. This difference was introduced to KC++ in order to make it easier to replace normal variables with futures. Assignment is a common operation with normal variables, so disallowing it in futures would have restricted their use.

Another difference between futures and references to a system-wide value is that a local copy of the value is kept in each active object (in fact, several copies may be kept if the same future is received several times). This can only create problems if copies of the value do not behave identically, which is possible if the value is an non-active object. For this reason it is important that if an object is used as a future value, then marshalling and unmarshalling it should produce an identically behaving copy. The KC++ parameter passing system relies on the same thing, so this is not a new restriction in KC++.

### 5.1.4 Creating futures

Future objects can be created in several ways. Probably the most common way is to get a future object as a return value from an active object method call. This is explained in detail in Section 5.2. Another way is to create futures from future source objects, which are discussed in Section 5.5. In both these cases the resulting future is probably pending, and it will get its final value later.

A third way to create a future object is simply to create it as a normal variable in the program. The future template class provides several constructors for this:

- **Future<T> f;** The default constructor creates a future which contains a value T(), i.e. the default value of the underlying type, created with its own default constructor. For example, `Future<int> if;` creates a future with value zero, and `Future<string> sf;` creates a future with an empty string as its value.

  This constructor can only be used if the underlying type has a default constructor, otherwise a compiler error is generated.

- **Future<T> f(/* parameters */);** If the future's constructor is given parameters, it creates a future which contains a value initialised with the same parameters. This way `Future<int> if(3);` creates a future with value 3. Similarly `Future<string> sf(10,'x');` creates a future which contains `string(10,'x')` — a string of ten "x"s.

  In the current KC++ this method of construction can be used if the underlying type's constructor needs at most 15 parameters. However, this limit could be easily increased, if necessary.
Future\(<T>\) \(f_2(f_1)\); (where \(f_1\) is another Future\(<T>\)) The copy constructor can be used to create a copy of another future of the same type. If the original future contains a value, the copy shares the same value, i.e. a copy of the value is not created. If the original future is pending, the new future becomes also pending. When the value becomes available, both the futures share the received value.

Sharing the value does not cause problems because the value inside a future cannot be modified. This way a programmer cannot accidentally change the value of one future by modifying another future sharing the same value. Sharing is automatically broken if either of the futures gets assigned a new value.

The constructors above make sure that a programmer cannot explicitly create “loose-ended” futures which would never get a value from anywhere. The first two constructor types create futures which already contain a value. The copy constructor creates a pending future only if the original future is already pending. If the original future is pending, it must have been originally created by an asynchronous method call or a future source, or it is itself a copy of such a future. If all methods eventually complete their execution, the return value futures eventually get their value. Similarly, if a future source gets eventually bound to a value or gets destroyed, all futures created from it get a value. This way futures created by the programmer cannot cause deadlocks by themselves, if the rest of the program works properly.

### 5.1.5 Assigning futures

Future types also have assignment operators which make it possible to assign a new value to a future. When a new value is assigned, the old value gets destroyed unless there are several futures sharing the same (possibly pending) value. In this case the assignment “detaches” the future from the shared value before assignment. Section 5.1.6 discusses the situation where a pending future value gets destroyed before receiving its value.

Future types have the following assignment operators (below, all objects with names beginning with \(f\) represent futures of type Future\(<T>\) and objects with names beginning with \(v\) represent values of type \(T\)):

- \(f = v\); When a simple value of the future’s underlying type is assigned to the future, the future discards its old value and creates a new value by storing a copy of \(v\). After the assignment the future is not pending regardless of whether it was pending before the assignment.

  It should be noted that the value \(v\) is not assigned using its own assignment operator, but rather a copy of it is created using the copy constructor. This is the only option as the semantics of KC++ does not allow changing an existing future value, making the use of assignment operator impossible.

- \(f_2 = f_1\); When a future is assigned to another future of the same type, the target of the assignment discards its old value and starts sharing the value of the other
future. If \( f1 \) is pending, \( f2 \) becomes pending too, and when the value becomes available, both futures share the same received value. This makes the behaviour of assignment compatible with the behaviour of the copy constructor.

Like the construction of new futures, assignment of futures makes sure that the programmer cannot end up with a pending future which would never receive any value.

Because assigned and copied futures share their value, future assignment and copy construction are very efficient operations. The underlying value does not have to be copied and the internal implementation of futures themselves is so simple that its assignment and copying does not take much time. Normal \( \text{C++} \) parameter passing uses the copy constructor, so passing futures by value inside an active object is also very efficient.

5.1.6  Destroying futures

When a future object gets destroyed, its destructor takes care of necessary actions. These actions depend on whether the future is sharing a value and on whether the future is still pending or not.

If the future to be destroyed shares its value (pending or not) with another future, it only detaches itself from the shared value. In this case no other actions are necessary.

If the future is not sharing its value with any other future and if it is not pending, its destructor destroys the future's value. If the value is an object, this includes calling the object's destructor.

The situation becomes more complex if the future is still pending and not sharing its value with other futures. In this case simply destroying the future would mean that there would not be any place where to receive the value when it finally becomes available. This would cause several kinds of problems:

- If the future's underlying type is a simple non-object type, it would not matter whether the future’s value would be received at all. The message containing the destroyed future’s value could simple be discarded by the \( \text{K\texttt{C++}} \) run-time library. The program could not in any way detect that the future’s value was not actually “received” at all.

However, futures’ underlying types may also be object types. In this case the construction of the future’s value (and its destruction) are events which may cause side effects outside the object itself. This means that the arrival of the future’s value may be detected by the program even when the actual future has been destroyed. For this reason the future’s value should be “received” and constructed properly even if the future itself no longer exists.
• If a pending future would simply be destroyed, it would mean that the thread owning the future would not expect the message containing the future’s value later when it arrives. This could be solved in the KC++ run-time library by simply discarding all messages whose recipient future does not exist. However, if all unidentified messages would be silently discarded, possible bugs in the KC++ implementation might go unnoticed.

• The KC++ system does not necessarily have to use message-passing mechanism to send values of futures from one active object to another. In a shared-memory system it is quite reasonable to use shared memory and semaphores to pass the future values between execution threads. In this kind of an environment it would be impossible (or at least disastrous) to simply destroy a pending future and its underlying semaphores and memory, as the sender of the future’s eventual value would still need to access the semaphore.

The KC++ run-time system solves all the problems mentioned above by keeping a list of all pending futures in each execution thread. This list is simply a list of actual future objects, each one sharing its value with a “real” pending future in the program.

When a future receives it’s value, the run-time system finds the correct pending future from the list, unmarshals the value inside the future and then destroys the future from the pending list. Because of value sharing, all appropriate futures in the program have automatically “received” the value. This sequence of events is shown in Figure 5.2 on the following page.

This mechanism makes sure that a future is never destroyed while it is pending. Every pending future in the program is always sharing its value — at least with a future in the pending list. If the actual future gets destroyed before its value is received, it detaches itself and leaves the future in the pending future list to wait for the value. When the value finally arrives, the KC++ run-time system finds the future in the pending list and unmarshals the value inside that future. After this the run-time system destroys the future from the pending list, but at this time the future is no longer pending and can be destroyed normally. This scenario is shown in Figure 5.3 on page 60.

The mechanism described above makes sure that all futures will eventually receive their value (causing the execution of the value objects constructor and destructor), even if the actual futures are destroyed before the value arrives. The run-time system also refuses to destroy an active object until all its pending future values have been received.

5.2 Futures as return values

The main use of futures is to allow asynchronous method calls with return values. If an active class has a method with return type T, the proxy class created by the KC++
CHAPTER 5. ASYNCHRONOUS METHOD CALLS AND FUTURES

There are three exceptions to this:

- Methods returning nothing (i.e. `void`) also return nothing in the proxy. Because there is no return value, no futures are needed. However, this also means that the caller cannot synchronise with the completion of such a method, i.e. `void`-returning methods cannot be called synchronously. If such a synchronisation is needed, the original method should return `Future<void>` (see the item below and Section 5.4).

- If the original method already explicitly returns a future type, this type is preserved in the proxy. This is sensible as following the normal rule would result in a “future future type,” which is seldom useful (but possible, nevertheless). The meaning of explicit future return types is explained in Section 5.2.3.

- If the return value is an active object or a pointer or reference to such an object,
CHAPTER 5. ASYNCHRONOUS METHOD CALLS AND FUTURES

![Diagram of Future Lifecycle](image)

**Figure 5.3:** Destruction of a future before receiving the value

no future is returned. However, the normal type changes from active classes to proxies, from pointers to KC++ pointer classes and from references to KC++ reference classes are performed. KC++ proxies, proxy pointers and proxy references are implemented internally using futures (see Section 6.2.2), so no explicit futures are needed.

Even though methods of the proxy return futures, the methods in the actual active object continue to return the same return values as before. Listing 5.2 on the following page shows a definition of an active class and the compiler-generated definitions of the corresponding proxy class and active class.

When an active object method is called through a proxy, the proxy creates a pending future object, sends its ID to the active object in the method message, and then returns the future object as the method’s return value.\(^1\) After receiving the

\(^1\)This is the only case in KC++ where a pending future originates in a different object than the object where its value is finally given.
Listing 5.2: Definitions of an active class and its proxy class

Original active class

```cpp
class Example : public Active
{
public:
    int giveInt();
    Future<double> giveDouble(); // (Section 5.2.3)
    Future<void> syncVoid(); // (Section 5.4)
    Example& operator=(const Example& e);
};
```

Generated proxy class

```cpp
class Proxy_Example :
{
public:
    Future<int> giveInt();
    Future<double> giveDouble();
    Future<void> syncVoid();
    Proxy_Example& operator=(const Proxy_Example&);
};
```

Generated active class

```cpp
class Example : public Active
{
public:
    int giveInt();
    Future<double> giveDouble(); // (Section 5.2.3)
    Future<void> syncVoid(); // (Section 5.4)
    Ref<Proxy_Example> operator=(
    const Ref<Proxy_Example>& e);
};
```
method message, the active object’s method invoker executes the method. When the method completes, the method invoker takes the return value and binds it to the return value future. After that the KC++ run-time system takes care of forwarding the value to the caller.

5.2.1 Asynchronous method calls

As mentioned before, all KC++ active object method calls are implemented as asynchronous calls, with proxies returning futures as return values. Therefore this section mainly discusses a few ways of utilising asynchrony and futures, as well as pointing out a few special cases.

In the simplest case the programmer simply calls an asynchronous method and then synchronises with the completion of the method later with wait (or polls the future with isready):

```java
Example e;
Future<int> fi = e.giveInt();
;
fi.wait(); // Synchronise
```

Alternatively the synchronisation may be implicit when the return value future is converted to its underlying type:

```java
void function(int i);
Example e;
Future<int> fi = e.giveInt();
;
function(fi); // Conversion & synchronisation
```

To benefit from asynchronous calls, the synchronisation should happen as late as possible, in order to minimise the time the caller has to wait for the actual return value to arrive. Assigning futures to each other and copying them does not require synchronisation, so futures should usually be used as much as possible. For example, in the previous example synchronisation happens before function is called. However, it is quite probable that this function does not use the value of its parameter immediately, so it could be sensible to change the parameter type to a future as well. Copying futures locally is very fast (see Section 5.1.4), so passing a future as a parameter instead of an integer does not usually add an unacceptable overhead to the program execution:

```java
void function(Future<int> i);
Example e;
Future<int> fi = e.giveInt();
function(fi); // No synchronisation
```
KC++ futures satisfy both the “CopyConstructible” and “Assignable” requirements, which ISO C++ containers require from their elements [ISO, 1998, Ch. 23]. Therefore it is safe to store futures in containers like vector or map. The speed of future copying and assignment make operations on such containers fast even if copying and assigning the return values themselves would be slow. No synchronisation is needed as long as the container operations do not need the actual value of the future. Listing 5.3 contains a function which creates (and immediately discards) ten Example objects, stores the return values in a vector, copies them to a list in reverse order, and finally synchronises with all the values using the function for_each in the ISO C++ algorithm library.

It is common in C and C++ to call a function returning a value without using the value, i.e. discarding the value completely. The C++ syntax is designed to allow this — the return value of a function or a method does not have to be assigned or copied anywhere. Doing this on futures poses no problems or dangers. The proxy method always returns a future (unless the method returns void). If this return value object is not used in the program, its lifetime ends automatically at the end of the full expression where the method was called (usually at the next semicolon). Then the return value future is destroyed without synchronisation as described in Section 5.1.6. The KC++ run-time system still knows about the future and receives the value of the future.

```
1 void example()
2 {
3     typedef Future<int> Fint;
4     vector<Fint> v;
5     for (unsigned int i=0; i<10; ++i)
6         { // Create new active object for each round
7             Example e; // Each active object is discarded at the end of round
8                 v.push_back(e.giveInt());
9             } // Insert elements to list in reverse order
10            copy(v.rbegin(), v.rend(), back_inserter(l));
11            copy(l.begin(), l.end(), ostream_iterator<int>(cout));
12        } // Synchronise with all elements and then display them
13    } // Insert elements to list in reverse order
14```

LISTING 5.3: Futures with ISO C++ containers
when it finally becomes available.

```java
Example e;
e.giveInt(); // Discards the return value
```

### 5.2.2 Synchronous method calls

Although all active object method calls are asynchronous, synchronous calls can be easily simulated by simply not using futures. The return value future returned by the proxy object is then implicitly converted to the original return value type, causing synchronisation immediately after the call:

```java
Example e;
int i = e.giveInt(); // “Pseudo-synchronous” call
```

The benefit of this “pseudo-synchronous” call technique is that it does not require any additional syntactic trickery when the method is called. It also makes the resulting code smaller as the KC++ compiler does not have to create different code for asynchronous and synchronous methods.

The simulated synchronous call does not cause any performance penalty either. Asynchronous call followed by synchronisation requires two messages between the objects: the method message to the active object and the future value message back to the caller. There is no way to accomplish a value-returning method call with less than two messages, even if a dedicated synchronous method call mechanism was implemented. The only additional overhead is the creation and destruction of future objects, but this overhead is likely to be small compared to the unavoidable overhead of passing information between two active objects.

There is one limitation caused by always asynchronous method calls. If a method returns nothing (i.e. `void`), the proxy method does not return anything either. Normally this is what is wanted, because there is no return value to synchronise to. However, sometimes a program may want to synchronise with the completion of a method, even if no return value is needed. Many future-based systems require that such methods return a “dummy” return value (a boolean value `true` is often used). In KC++ such dummy return values are not needed, because the designer of the active class can write methods with return type `Future<void>`. This explicit future return value allows the caller to synchronise with the completion of the method without actually receiving any “value.” These void-futures are explained in Section 5.4.

### 5.2.3 Explicit future return types

When a future is received as a return value from an active object method call, it normally receives its value when the method completes its execution. In this case the caller always synchronises with the completion of the method. In many situations
this is too restrictive. Clearly the synchronisation cannot occur before the method completes, as there is no return value until that point. However, it may be useful to synchronise to some event which happens after the method has been completed.

For example, in an active object representing an empty buffer, the method get might return a future which does not get its value when the method completes, but only after the put method has been used to add an element into the buffer. To achieve this kind of functionality, an active object method must be able to return a value which is not necessarily known when the method is completed.

The KC++ system solves this by allowing explicit use of futures as return types in active classes. When an active object method return type is not a future, the corresponding proxy class returns a future for that type, and the future gets its value when the method completes its execution. However, if the return type of an active object method is a future, the method in the proxy class simply returns this same future to the caller (and not a future to a future).

This mechanism allows the active object to return a future as its return value. This future is then simply passed on to the caller. In this case the caller has no way of knowing when the method itself has been completed, as the value of the future may or may not be known at that time. Later, when the value of the returned future becomes available to the active object, the KC++ run-time system automatically forwards the value to the caller.

Explicit return of futures is a very useful mechanism when it is combined with the ability to explicitly create pending futures and later give them value using future sources (which are explained in Section 5.5). In the buffer example the empty buffer object might create and store a future source for each “unavailable” value it returns with get, and return a pending future bound to that future source as a return value. Later, when put method is used to add a value to the buffer, it would bind the value to the future source, automatically causing the run-time system to forward the value to the caller of get. This kind of buffer is presented in Appendix C.

Another use for explicit future return types is to allow synchronisation to an other method in a different active object. If a method returning an explicit future returns a future that is the return value of another method call, this future is forwarded to the caller and the original method completes its execution. This results in a future which receives its value when the other method (called by the original method) is completed. Listing 5.4 on the following page contains a piece of code which demonstrates forwarding of return values.

If the operation described in the previous paragraph is tried in a method which does not have an explicit future return type, the situation is different. In that case the original method waits for the called method to complete before it itself completes its execution. This blocks the active object from other calls, because there can only be one uncompleted method in each active object at any time. Listing 5.5 on page 67 shows the code in Listing 5.4 without an explicit future return type. This makes
explicit future return types a convenient way of preventing methods from blocking active objects unnecessarily.

Even when explicit future return types are used, the active object can still return a normal non-future value. There is an implicit type conversion from the underlying type of the future to the future type itself, so returning a non-future simply creates a temporary non-pending future object, and this future (with its value) is then forwarded to the caller, causing the caller’s return value future to receive its value when the method completes.

### 5.3 Futures as parameters

Futures are useful for storing return values of asynchronous method calls and synchronising with them, but they have broader uses as well. They can be thought as “synchronisation tokens” which can be passed on to allow others to synchronise with selected events. This requires that futures may be passed between active objects as parameters. C++ futures allow this as long as the underlying type is acceptable as an active object method parameter.

Return values are not the only way to get information back from a method in C++.
class NonExplicit : public Active
{
public:
int calculate();
private:
Helper h;
};

int NonExplicit::calculate()
{
// Do some other calculations
return h.nextValue(); // Synchronise first, then return value
}

LISTING 5.5: Listing 5.4 with a non-future return type

They are sufficient only when there is a single value which has be returned. Although sometimes data structures like “pair” are used to return multiple return values, it is much more common to get additional return values through reference parameters. Although references are not usually accepted as KC++ active object method parameters (see Section 4.2.6), future references are an exception, so that several return values can be returned.

5.3.1 Futures as value parameters

When a future is passed as a parameter to an active object method which does not explicitly request a future parameter, the situation is simple: an implicit type conversion is used to convert the future to its underlying type, and this conversion waits until the value becomes available. The value is then passed as a normal value parameter.

However, if the method explicitly takes a future type as its parameter, the situation is more complex. The way parameter passing happens depends on whether the future already contains a value or whether it is still pending.

If the the value of the future is available, the parameter passing resembles the non-future case described earlier. The method message contains simply a boolean flag indicating that the future is not pending and the value of the future. When a copy of the future is recreated on the active object side, this future also contains a copy of the value.

If the future is still pending when it is passed as a parameter, the method message only contains a boolean flag indicating a pending future and an ID field identifying the pending future. The copy of the future created on the active object side is also pending, ready to receive the value from the caller.
Later, when the value becomes known to the caller, the KC++ run-time system automatically forwards the value to the active object, which stores the value in its local future. This way passing a pending future results in two messages to the active object — the actual method message and later a message containing the value of the future.

5.3.2 Futures as reference parameters

Reference parameters are normally used for three purposes in C++. Constant references are used as “in” parameters to pass read-only values to methods without copying them. On the other hand, non-constant references are used for “in-out” parameters, i.e. to pass a value to the method and later receiving a value back. Another use for non-constant references is as “out” parameters, where the programmer does not actually pass any value to the method, but is only interested in the value of the parameter after the method completes.

Passing non-active parameters between active objects always involves copying the parameter values because each active object can reside in its own address space. Therefore it is not possible to achieve the effect of constant reference parameters, where the lack of copying is usually the only reason they are used. For this reason KC++ does not allow constant future reference parameters in active object methods.

Non-constant future references can be used for either “in-out” or “out” parameters. The KC++ compiler has no way to know which use is meant, so it uses the “in-out” assumption which also includes the “out” case. When a future is passed as a reference parameter to another active object, the code in the proxy object first inserts the current value of the future (pending or not) into the method message, just like with normal future parameters. Next the proxy detaches the future from its old value and binds it to a new, pending value. The ID of this value is also included in the method message.

When the method is invoked in the active object, it creates a future with a value that was included in the method message. A reference to this future is then passed to the actual method. When the method is completed, the value of the parameter future (either pending or not) is sent back to the caller in the same message as the return value of the method. The ID received in the method message is used to indicate that the value should be bound to the future reference parameter.

This way the value of a future reference parameter becomes known at the same time as the return value, meaning that reference parameters can be used to replace return values without changing the semantics of the program. It should be noted that future reference parameters are similar to explicit future return types in the sense that they can be used to forward pending futures received elsewhere in the program.

If a future reference parameter is used as an “out” parameter, the mechanism described above is usually quite appropriate. If the program creates a new future and then uses it as a reference parameter to receive a value from a method, the value of the future in the method message is never used in the active object. A little more prob-
lematic situation occurs if a pending future is passed as an “out” parameter this way. In this case the eventual value of the parameter is actually unnecessary. However, the parameter passing mechanism sends the pending future to the active object, even if its value is never used. When the value becomes known, the KC++ run-time system forwards the value to the active object, even if the method has already completed. This is necessary because the run-time system cannot know whether the active object needs the value of the parameter or not.

A similar situation arises if a pending future is passed as a future reference parameter, and the active object never uses the parameter. The KC++ run-time system in the caller cannot know this, so it considers the value received from the method to be a different value than the old one. If the old value of the future is received only after the method has completed, the run-time system nevertheless sends the value to the active object, whose run-time system simply binds this value to the “new” value of the parameter future and sends the value back to the caller. This “detour” cannot be avoided because the active object and the caller have no information on each other’s actions.

5.4 Pure synchronisation — void-futures

Although futures are normally used as place-holders for asynchronous method return values, they have another important property — they allow callers to synchronise with the completion of a method. Synchronisation in general is separate from return value passing (although the act of receiving a return value always includes synchronisation).

Despite being originally designed as place-holders for a value, futures can also be used for pure synchronisation. The C++ language has the type void, which is a “non-entity” in the sense that it represents a type which can contain nothing, i.e. an “empty type” with no values. Variables of type void cannot be created in C++, but otherwise void is a completely normal type.

The KC++ systems allows the creation of futures with void as the underlying type. This future type, Future<void> is a specialisation of the normal future template. It contains all the necessary mechanisms for synchronisation, but no value. This means that void-futures have methods isready and wait, but not the method value or an implicit type conversion (the missing methods would be useless anyway, because value would return void, and type conversion to void is practically an oxymoron).

Normally there is no way to synchronise with the completion of a method returning void. If such synchronisation is wanted, it is easy to implement with void-futures. The method can explicitly return a void-future, and the caller can wait using this future. The method itself can then create a non-pending void-future and return that as its return value, making the void-future on the caller’s side also non-pending and releasing the caller. Listing 5.6 on the following page shows an example object with this kind of a method.
A method returning a void-future can be called like a method returning nothing, like on line 16. In this case the call is asynchronous and the void-future return value is discarded (the KC++ run-time system takes care of the eventual return value message). If immediate synchronisation is wanted, the caller can directly call wait on the returned void-future, like on line 17. Like with normal futures, the third possibility is to occasionally poll the void-future with isready and perform some other actions while waiting. This is shown on lines 18–19.

It would have been possible to design KC++ in a more consistent way and make proxy methods return void-futures if the actual method returns void. This would have made it possible to synchronise with any method. However, most of the time the caller of a method returning nothing is not interested in synchronisation. Automatically returning a void-future would have meant that the active object would have notified the caller also in these cases, adding overhead to the system. For this reason it was decided that void-futures would be available only as explicit future return types, not automatically.

In addition to synchronisation with methods with no return value, void-futures are very useful when combined with future sources explained in Section 5.5. They can be used as explicit synchronisation tokens which can be stored in containers,
passed between active objects etc. They also make KC++ simpler because futures can be used as the only synchronisation mechanism in the whole system.

5.5 Creating futures explicitly — future sources

Return value futures and asynchronous method calls alone are not enough for concurrent programming. If return value futures were the only mechanism for synchronisation between active objects, all synchronisation would be restricted to the completion of some method. Explicit future return values allow this method to be different than the one the programmer calls directly, but still each future would receive its value at the end of some method call.

The restriction comes from the fact that all active objects in KC++ execute their methods with one-method-run-to-completion semantics. If one method call is still in progress, the execution of the other methods is started only after the current method is completed. This makes it impossible to return a future from one method call and give it a value as a result of another method call.

For example, it could be reasonable to implement the barrier synchronisation mechanism [Andrews, 1991, Ch. 3] as an active object. A barrier is a mutual synchronisation point for several execution threads. Each thread calls an operation “commit”, which blocks the execution of the thread. When a predetermined number of threads are waiting in the barrier, they are all released simultaneously. In KC++, the commit-operation could return a void-future for synchronisation with the barrier. Listing 5.7 on the next page shows an outline of such a barrier class.

This barrier class cannot be implemented using only normal return value futures. The commit-operation has to return a future to the caller. If this future was created in the method itself, the caller would proceed immediately after the method completes. The method itself cannot wait for other participants, as the calls of the other participants are blocked until the method completes. The only alternative would be to get the return value future from another active object, which would perform the actual barrier synchronisation. However, exactly the same limitations apply to this active object too, so it cannot create the future either. As a result, the barrier class is impossible to implement this way.

The KC++‘s solution the problem are future sources. Future sources are objects which can be used to explicitly create pending futures. Later the same future source object is used to give a value to all futures which have been created from that particular future source — or any future created from these futures. This way future sources separate the actions of future creation and value binding from each other. They make it possible to return a pending future from one method and later give it a value in another method.

Like futures, future sources are a class template with the underlying type as a type parameter. Futures with the same underlying type can be created from a future source
CHAPTER 5. ASYNCHRONOUS METHOD CALLS AND FUTURES

```
class Barrier : public Active
{
public:
    Barrier(int newcount);
    Future<void> commit();
private:
    const unsigned int target_count;
    unsigned int count;
};

Barrier::Barrier(int newcount)
    : target_count(newcount), count(0) // ...
{
}

Future<void> Barrier::commit()
{
    if (++count == target_count)
    {
        // Release the future if enough participants
    }
    // Return the future for waiting & synchronisation
    return sync;
}

// An example object using the barrier
class Process : public Active
{
public:
    Process(Barrier& newbarrier) : barrier(newbarrier) {}
    void run()
    {
        // Do something
        barrier.commit().wait(); // Synchronise with others
        // Continue
    }
private:
    Barrier& barrier;
};
```

LISTING 5.7: Outline of an active class implementing a barrier
object through an implicit type conversion. When a future source object is created, it is initially “empty,” and does not contain any value. All futures created from this kind of empty future source are pending futures without any value either.

In addition to the type conversion for creating futures, future sources have a method called bind. This method can be used to “fill” the future source and give it a value. It creates an object of the underlying type, using its parameters as constructor parameters. After this, all futures which have been created from this future source are bound to this value. The KC++ future forwarding mechanism is used to send the value to futures in other active objects, if necessary. The act of binding the future source to a given value is similar to giving the return value future its value at the end of an active object method.

After the future source is given its value using bind, all subsequent futures created from it contain this value and are not pending. In the current version of KC++ the bind method can take up to fifteen constructor parameters, as long as the underlying type has a constructor which can accept the parameters. If necessary, it is very easy to extend this limit. Another way to overcome this limitation is to create an object with the correct value outside the future source and then give this object as a parameter to bind, causing the future source’s value to be created using the copy constructor.

The barrier problem is now easily solved using future sources. Listing 5.8 on the following page shows the complete barrier class. It contains a void-future source as a data member. This future source is used to create futures sent to the participants. Every time a new participant calls the commit method, counter is incremented. If there are too few participants, a pending future is created from the future source and returned to the participant on line 29. When the final participant calls commit, the method first executes bind on the future source, releasing all participants waiting for their futures to get a value (or polling them). The underlying type of the future source is void, so there is no real “value” created here, but the synchronisation caused by binding is still performed. Finally the method creates a non-pending future on line 29, so the final participant does not have to wait.

If all goes well in the example, the participants are finally released through the futures. However, it is possible that the barrier object gets destroyed before enough participants have called commit. Although this probably should never happen, it could be possible because of exceptions etc. If a future source is destroyed before it is bound to a value, it is automatically bound during its destruction. There are no parameters to create the value from, so the default constructor of the underlying type is used. This makes sure that the destruction of a future source does not leave “stranded” pending futures.

Future sources can also be copied and assigned to each other. In this respect they act like futures. When a future source is copied, both future source objects share the same “emptiness” (or value if the original future source is already bound), so that futures created from either future source receive their value when bind is called on either of the future sources. When a future source is assigned to another future
class Barrier : public Active
{
public:
    Barrier(int newcount);
    Future<void> commit();
private:
    const unsigned int target_count;
    unsigned int count;
    SFuture<void> sync;
};

Barrier::Barrier(int newcount)
    : target_count(newcount), count(0), sync() {
}

Future<void> Barrier::commit()
{
    if (++count == target_count) {
        // Release the future if enough participants
        sync.bind();
    }
    // Return the future for waiting & synchronisation
    return sync;
}

LISTING 5.8: The barrier class using a future source

source, the target of the assignment first binds itself to the default value and then
starts sharing the value of the other future source. In other words, assignment follows
the common destroy-then-copy semantics.

The barriers in Listing 5.8 can only be used once. However, the future source sync
can easily be “reset” to non-bound state by assigning a new empty future source to it. Listing 5.9 on the next page shows an improved barrier class which resets itself
every time enough participants have called commit. The barrier counts the number of “rounds” the barrier has made and returns this number as the return value of commit.
If the barrier object is destroyed prematurely, the participants which have already
committed themselves receive value zero (default value of unsigned integers) as a
round number.

In addition to resetting future sources, assignment and copying make it also possible to store future sources in ISO C++ containers like vector or map. Appendix B,
CHAPTER 5. ASYNCHRONOUS METHOD CALLS AND FUTURES

contains an example implementation of a priority semaphore as an active object. The
semaphore uses a ISO C++ container priority_queue to release waiters in priority or-
der.

Each future source can be thought to represent a source for a single value which
becomes known later in the program. Therefore it is important that each future source
is bound to a value only once, i.e. that an already bound future source is not “re-
bound.” This restriction becomes very important if future sources are copied and
assigned, in which case a future source can become bound through another future
source. The bind-once restriction cannot be verified at compile-time, so a dynamic
check is necessary. If an attempt is made to bind an already bound future source, the
method bind throws an exception bad_bind. Futures sources also provide a method
isbound which can be used to check the state of a future source.
5.6 Locking active objects

Methods of an active object do not necessarily provide the correct granularity for using the object. Often a “service” provided by an active object consists of a series of method calls, and the caller may need to perform some processing between the method calls. In many cases it is important that other threads do not access the object between the method calls of the “service” or “transaction.” In other words, it must be possible to lock objects for the duration of several method calls.

It is very important that each lock is eventually released. This becomes easily complicated when C++ exceptions are used, because exceptions create new possible flows of control in the code, and the locks have to be released in all of them. A useful C++ idiom for providing exception safe locking are guard objects, also known as “scoped locking” [Schmidt, 1999]. A guard object is an object whose constructor acquires the necessary lock and whose destructor releases the lock. When a guard object is created in a scope, the lock is in effect after the creation of the object. When the scope ends, the guard is destroyed and the lock automatically released. This happens even if the scope is left because of an exception.

The KC++ system uses this idiom with a guard class ActiveLock. When an object of this class is created, the constructor of the object sends a message to the active object it receives as a parameter, requesting it to grant a lock. If the active object is not already locked, it locks itself and signals the lock object. If the active object is already locked, the lock request is not processed until the old lock is released. The lock requests enter the same method queue as other active object methods, so they are serviced in FIFO order.

After the lock object constructor gets confirmation that the active object is locked for exclusive access, it completes and lets the thread continue its execution. After this the code can use the active object normally, without other users getting in the way. If the locker tries to lock the object again while it already holds a lock, the later lock request has no effect. However, the KC++ system keeps a count of these lock requests, so that the object is released only after the same number of release requests has been issued.

When the program exits the scope of the lock object (either normally or because of an exception), the destructor of the lock object sends a release message to the active object. The object then releases the object lock and continues to process method requests from any caller. If other users have called methods while the lock was in effect, these method requests are stored in the method queue and are executed after the lock is released. These delayed requests can also include further lock requests from other threads.

Lock objects provide a very natural way to limit the duration of locking to a certain portion of the code. Listing 5.10 on the following page shows a simple function which uses object locking. Lock objects can also be used as data members of an object, in which case the active object is locked for the lifetime of the object containing the lock object.
class Buffer : public Active
{
public:
  void put(int number);
  int get();
  bool empty() const;
};

int getIfPossible(Buffer& buffer)
{
  // Get a number if buffer not empty, otherwise return 0
  Future<int> number(0); // Return value if empty
  { // Create scope for locking
    ActiveLock lock(buffer); // Lock the buffer
    if (!buffer.empty()) { number = buffer.get(); }
  } // Closing the scope releases the lock
  return number; // Wait for the number and return it
}

void putTwo(Buffer& buffer, int i1, int i2)
{
  ActiveLock lock(buffer); // Lock the buffer
  buffer.put(i1);
  buffer.put(i2); // Store the two number next to each other
  // Lock is automatically released
}

LISTING 5.10: Object locking using lock objects

Lock objects can also be combined with standard C++ auto_ptrs to safely lock an object in one place and release it in another. In this case the lock object is created dynamically with new and then bound to an auto_ptr which implements strict ownership semantics [Kreft and Langer, 1998]. This auto_ptr can then be passed as a "lock token" to another function etc. The auto_ptrs pass the ownership of the lock object to each other and make sure that the lock object is automatically destroyed (and the lock released) when the last auto_ptr is destroyed. Listing 5.11 on the next page shows an example of this.

The class Active has also methods lockActive and unlockActive. These methods can be used if explicit locking without lock objects is needed. This is normally not recommended, because handling object locking and unlocking with separate methods calls causes a deadlock if a user forgets to release an object lock. C++ is especially prone to these kinds of errors because of the exception handling mechanism. If an
exception is thrown while an active object is locked, the program execution may skip the code containing the lock release call. To prevent this the programmer would have to design his error handling code carefully, taking into account every possible way a lock might be left unreleased.

One use for `lockActive` and `unlockActive` methods is that they can be called from other methods of the active object. This makes it possible to write methods which, once executed, lock the object exclusively for the same caller. Similarly another method could then release the lock at the end.

Object locks increase the possibility of deadlocks, even if lock objects are used. For this reason they should not be used unless locking is really necessary.
Chapter 6

Implementation of KC++

This chapter discusses the current implementation of the KC++ system. The system consists of two parts: the KC++ compiler and the KC++ library. One of the design principles of KC++ has been to keep the compiler as simple as possible and implement most things in the library. There are two benefits from this approach:

- The KC++ compiler is based on an existing C++ compiler front end. Keeping the compiler as simple as possible makes it easier to change the compiler to use a different C++ compiler front end, if necessary.

- Debuggability has been another important design issue in KC++. This means that the source code produced by the compiler should be as close to the original source code as possible. If some feature can be implemented in the KC++ library, it means that the compiler does not have to alter the original source code to implement that feature.

The first section in this chapter describes the KC++ compiler and the changes it makes to the original source code. The second section explains the structure of the KC++ library and discusses some design issues related to the library implementation.

6.1 The KC++ compiler

The current KC++ compiler is implemented as a back end to an EDG C++ compiler front end from Edison Design Group [EDG, 1999]. The KC++ syntax is exactly the same as normal C++ syntax, so the compiler front end has not been modified in any way. The interface between the EDG front end and the KC++ back end has been kept as small as possible, and it has been isolated into its own program module. This means that coupling between the front end and the back end is relatively low, making it easier to implement a KC++ compiler using a different C++ front end, if necessary. It also makes it possible to publish most of the KC++ compiler program code without violating non-disclosure agreements with Edison Design Group.
The compilation process is shown in Figure 6.1. During the compilation the compiler front end first compiles the KC++ program into its own intermediate code, which is represented as a graph-like data structure in the compiler’s memory. The C++ syntax checking is performed during this phase, so only those KC++ programs which are syntactically correct C++ enter the second phase.

During the second phase the KC++ back end scans through the intermediate code to find all code related to active objects. It then uses the original source files to create new, modified C++ source files. These files are then compiled with a conventional C++ compiler. The g++ compiler from the Gnu Compiler Collection (GCC) [GNU, 2000] has been used for this purpose, but any C++ compiler should do. The source generated by the back end is standard-conforming and uses the KC++ library to achieve concurrency. The compiler-generated source files are created in their own directory with the same names as the original files.

**Figure 6.1:** The KC++ compilation process
Finally the resulting object files are linked with the KC++ library, resulting in the final executable programs. The current KC++ system only creates one executable, but it is also possible to make a separate “server” executable for each active object. This could be useful in a distributed environment, where each active object is run on a different computer.

The modifications to the original source code are kept at minimum. Using modified original source code instead of generating the code from the intermediate code makes the resulting code very close to the original. Because of this the source code generated by the KC++ compiler is completely human-readable and also mostly understandable to programmers who are not familiar with the internals of KC++.

The high correlation between the original and generated source also helps in debugging KC++ programs using normal C++ debuggers. The line numbers in the generated code are the same as in the original source, except for files containing definitions of active classes, where an additional line has to be added to each definition, causing an offset of one line per active class. However, \#line directives are added to necessary places in the generated source so that debuggers and other code analysers still refer to the correct lines in the original source code.

Each C++ source file generated from the original KC++ source has a \#line directive as its first line. This directive links the generated file with the original source file. It informs the actual C++ compiler to refer to the original source code in the debugging information it adds to the object files. For example, if the original file was “test.cc,” then the compiler-generated file “kcpp/test.cc” would start with the line

1 \#line 1 ".:/test.cc"

The rest of this section describes briefly the changes made to the original KC++ source when the compiler generates the resulting C++ source.

### 6.1.1 Active class definitions

When the KC++ compiler encounters a definition of an active class (i.e. a definition of a class derived from Active), it creates the corresponding proxy class. This class is defined in its own header file, created by the compiler. The compiler also creates a source file which contains definitions of the proxy’s member functions.

As explained in Section 3.2.3, proxy classes form an inheritance hierarchy similar to that of the active objects, with the base class Proxy on top of the hierarchy. The proxy class contains future-returning versions of the methods of the active class. The definitions of these methods create method messages, send them to the active object, and return appropriate futures. Constructors of the proxy class use the KC++ library to create a new thread of execution which then starts executing the method invoker of the new active object. Most of the functionality in the proxy class is written using services of the Proxy base class. This keeps the amount of generated code at minimum and helps to make the executable smaller.
To be able to use the proxy class, the generated source file must also include its definition. To achieve this, the compiler generates an `#include` statement before the definition of the active class. This statement reads in the generated proxy header file. The statement has to be on a line of its own, so an additional `#line` directive is added after it so that the following lines refer to the correct lines in the original source. Listing 6.1 on the next page shows an example of these directives. The inclusion of the proxy header is the only place in the generated source where the line numbering of the original file has to be modified.

When the definition of an active class is handled, the KC++ compiler also creates a source file containing the code for the active object method invoker. This code simply receives method messages, unmarshals the parameters and calls the appropriate methods of the active object. It is also responsible for creating and destroying the actual active object. The invoker is implemented as a static member function of the active class, and the definition of the active class is modified to contain the declaration of the invoker.

### 6.1.2 Uses of active objects, pointers and references

All the code which uses active objects must be modified to use proxy objects instead. The KC++ compiler accomplishes this by scanning the intermediate code for any uses of active class names. It then replaces these with the names of appropriate proxy classes. In the previous example all occurrences of `Testclass` are replaced with `Proxy_Testclass`. There is one exception to this: the active class name is kept intact in the qualifications of the active class’s member function definitions and in the active class definition itself. This means that the definition of member function `foo` of `Testclass` would still begin with `Testclass::foo`. This is of course necessary because the member function definitions really are members of the active class, not the proxy.

In addition to this, all active object pointers and references have to be changed to instances of proxy pointer and reference templates for reasons explained in Section 4.3. In the example `Testclass*` is replaced with `Ptr<Proxy_Testclass>` and `Testclass&` with `Ref<Proxy_Testclass>` everywhere in the code.

The changes above do not affect the syntactical correctness of the program. The pointer template defines all the same operations as a normal pointer, except for pointer arithmetic. However, arrays of active objects are not allowed, so pointer arithmetic is useless, anyway. Proxy classes define their own `&`-operator which returns a KC++ pointer object, so expressions like

```cpp
Proxy_Testclass o; // Originally Testclass o;
Ptr<Proxy_Testclass> p = &o; // Originally Testclass* p = &o;
```

still compile. The pointer template class has a constructor and an assignment operator which accept normal C++ pointers. These are used when a pointer object is initialised or assigned from a `new` expression which has created a new proxy object (new of course still returns a normal C++ pointer).
CHAPTER 6. IMPLEMENTATION OF KC++

LISTING 6.1: Compiled active class definitions
The KC++ library derives the reference template from its type parameter (the proxy class), so it automatically contains all the public member functions of the proxy. The derivation also makes it possible to use reference objects everywhere where proxy objects are expected. The reference template class has a constructor which allows creating a reference object from a proxy object, so proxy objects can also be used to initialise a reference as in normal C++. Section 6.2.2 discusses the structure of pointer and reference templates in more detail.

6.1.3 Additional changes to pointers

There are a few more modifications needed to make pointer objects behave correctly. The KC++ back end scans the intermediate code for any places where the integer constant zero is used as an active object null pointer. These occurrences of zero are surrounded with an explicit type conversion, because implicit type conversions do not work everywhere. This results in a code like

```cpp
// Originally Testclass* p = 0;
Ptr<Proxy_Testclass> p = Ptr<Proxy_Testclass>(0);
int i = 0; // Not changed, not a null pointer constant
```

When a dynamically created proxy object is destroyed using `delete`, the code has to be modified because `delete` needs a pointer, and the KC++ pointer object is not a pointer. This is solved by replacing all expressions `delete` p (where p is a pointer to active object) with `p.destroy()`, where the `destroy` member function of the pointer template class takes care of deleting the dynamically created proxy object.

Each object in C++ has an implicitly defined pointer `this` pointing to the object itself. Because the pointer is implicitly defined, it remains a normal C++ pointer in an active class. However, it too should be changed to a pointer object in the produced code. This problem is solved by replacing each occurrence of `this` with a call to the pointer class’s static member function `Ptrthis()`. This function creates a new proxy object pointing to the active object itself and then returns a pointer object pointing to this proxy. Listing 6.2 on the following page shows an example member function of `Testclass` before and after the modification.

6.2 The KC++ library

The KC++ library contains all the KC++ code which is not compiler-generated. As much of the KC++ as possible has been implemented in the library, because making changes to the library is easier than changing the KC++ compiler and because keeping the compiler small makes it more portable.

The library also contains all the functions which are responsible for creating new active objects (i.e. new threads of execution) and communicating between them.
This means that if someone wants to change these aspects of KC++ (to optimise KC++ for shared-memory processors, for example), then only the KC++ library has to be changed.

This section describes the overall structure of the main parts of the KC++ library: base classes Proxy and Active, active object pointer and reference template classes, future classes, and message streams.

### 6.2.1 Implementation of the base class Active

Each active objects in KC++ has to be derived from the base class Active. Mainly this base class acts as a “tag” which informs the KC++ compiler that all classes derived from it are active objects. The compiler then knows to write appropriate proxy classes and modify the program code to use these proxies.

The base class Active is not merely a tag, however. It also contains methods common to all active objects. At the moment this only means the methods lockActive and unlockActive needed for explicit object locking. Active objects can also use the method getlocker to ask for the ID of the process which has locked the object.

Internally the class contains information about the ID of the process which has invoked the current method, the ID of the process which has locked the object (if any), and the current lock count (this is used to track the situation where the same process “relocks” the object when it already holds a lock).

### 6.2.2 Implementation of proxies, pointers and references

Just as each active class is derived from Active (either directly or indirectly), each corresponding proxy class is directly or indirectly derived from the base class Proxy. This class defines most functionality needed in the proxy classes: method parameter handling, return value future handling, proxy marshalling and unmarshalling, and object locking.
Internally each proxy object only needs to contain enough information to identify the active object it represents. The type of this data is the same for each proxy object regardless of its actual class. Therefore the base class `Proxy` contains this information and the derived proxy classes do not contain any data members of their own.

As the class `Proxy` provides all the necessary functionality for proxies, actual compiler-generated proxy methods are very simple. They mainly call methods of the base class to create the method message, set up the return future and send the method message to the active object. Listing 6.3 shows the implementation of the proxy method `testmethod`. First a method message is created and the method ID (a string in the current test implementation of `KC++`) is added to the message. Then a value parameter and a return value future are added to the method message using appropriate helper methods. Finally the method message is sent and the method returns the future.

References to proxies are implemented as a simple template class in the `KC++` library. Each reference is derived from the proxy class it refers to, which automatically gives the reference object the same public interface as in the actual proxy. This derivation is implemented by defining the reference template in the following way:

```cpp
template<typename T>
class Ref : public T {
    ...
};
```

The only difference between a proxy reference object and a real proxy object is that the public constructors of the real proxy class create a new active object, whereas the constructors of the reference class simply bind the reference to an existing active object.

Pointers to proxies are also implemented as a template class. Because most of the functionality in different pointer types is exactly the same, the `KC++` library defines

```cpp
Future<int> ProxyTestClass::testmethod(double p1) {
    ObjectOMsg omsg;
    _setupMethod(omsg, "testmethod:int(double)");
    _add_valpar(omsg, p1);
    Future<int> retval(_add_retval<int>(omsg));
    omsg.send();
    return retval;
}
```

LISTING 6.3: A compiler-generated proxy method
a common base class for all proxy pointer types. This class contains all the common code, which is thus automatically shared between different proxy pointers. The actual pointer template implements only the type-specific part of the interface. The common base class also makes it easier to implement pointers in a way which makes it possible to assign a derived class proxy pointer to a base class proxy pointer, for example. However, the details of the pointer implementation are beyond the scope of this thesis.

Active objects (proxies) and pointers and references to them can be used as active object method return values. Because all method calls are asynchronous, the proxy, pointer and reference classes have to provide future-like functionality. This is embedded inside the classes, so that proxy objects, as well as pointer and reference objects, can also be “pending,” i.e. not yet bound to any active object. This state is hidden from the user, and if the proxy, pointer or reference object is used before it has its value, the program execution is simply paused until the value becomes available. The implementation of proxies, pointers and references as futures also makes it possible to share quite a lot of their code with futures, reducing the size of the KC++ library.

6.2.3 Implementation of futures

The KC++ futures are implemented completely as a library template class and do not require any support from the KC++ compiler. As with proxies and proxy pointers, all KC++ future types have a common base class which contains practically all the operations. This base class is not a template, so all futures share the same code, and introducing new future types to the program does not significantly increase the size of the program. Future sources are also implemented using the same class hierarchy, so they also share much of the same code.

Normal futures

Several future objects may share the same (possibly pending) value. This means that the value itself cannot be directly stored in a data member of a future, because it would then be automatically destroyed when this particular future is destroyed. To allow sharing the KC++ futures use the “pimpl” [Sutter, 2000] idiom and store all the necessary data in a private class. Each future object only contains a pointer to this private object, which contains the actual value of the future and other necessary synchronisation information.

This “value object” has a reference counter, which is incremented each time a new future starts sharing the value with an old one. Correspondingly the counter is decremented when a future object is destroyed and thus detached from the structure. This implementation makes futures very light-weight. Copying and assigning a future only requires updating the reference counts. This method is usually known as the
"counted handle/body" idiom [Coplien, 1992] (or [Coplien, 2000]).

The internal class has to be different for each future type instantiated from the future template, because the type of the value the future contains is different. However, all other parts of the value objects are identical. The KC++ futures utilise this fact by using the “Rungs of a Dual Hierarchy” pattern [Martin, 1997]. The mutual base class of futures defines a “value base class.” This class only contains the reference count and synchronisation information.

The future template then derives a new template class from this value base class. This derived template class has an additional data member of the correct type to contain the actual value of the future. Figure 6.2 shows the structure of the KC++ future implementation. It has been somewhat simplified to make the figure clearer.

The void-future class is a template specialisation of the actual future template, because it differs from other futures. The class Future<void> does not provide any member functions for accessing its value because it does not contain any value. Similarly the “value class” of the void-future is a specialisation of the general value class template, and contains no actual value member.

The value base class also defines pure virtual functions for marshalling and unmarshalling. The derived class template then defines these functions appropriately so that they marshal and unmarshal the future value. This allows the code in the future base class to perform marshalling and unmarshalling without knowing the type of the future. Because of this the future classes themselves do not have any virtual functions. This makes the future objects even more light-weight, because they do not have to contain virtual table pointers and other overhead related to dynamic binding.

**Future sources**

Future sources are implemented by deriving the future source template from the normal future template using private inheritance. The use of private inheritance is appropriate here, because future sources “are implemented in terms of” [Meyers, 1998,}

---

**Figure 6.2:** The KC++ future implementation
Item 42] normal futures. Future sources could not have been implemented using aggregation, because they have to have access to the protected member functions of the future class.

Deriving future sources from futures makes their implementation easy. For each future source there is now automatically a local future value object (in the future class). When new normal futures are created from the future source, they simply start sharing the same value object. When the future source is bound to a value, it stores the value in the value object. This also makes the value available to all normal futures sharing the same value object. Furthermore, the future value forwarding system (explained in the next section) then automatically sends the value to necessary other active objects.

Several future sources may share the same pending value. Therefore the future value class also maintains a reference count of future sources sharing the same value object. This reference count is used to notice if all future sources are destroyed without binding any value. If this happens, the last future source to detach itself from the value object automatically binds the object to the default value of the underlying type.

**Forwarding future values**

The future value forwarding mechanism is a very important part of KC++ because, in addition to parameter passing in method messages, it is the only mechanism for sending data from one active object to another. There are two sides to future forwarding: the sender side and the receiver side. Both sides have to know about the situation when a future is pending — the sender has to send the value to the correct destination, and the receiver has to know where to store the received value.

Each future in the whole system has a unique *future ID*. This ID identifies the active object where the future resides as well as the future itself inside the active object. These IDs are used as “addresses” or “tags” when sending future values.

On the receiver side, a future may become pending in three different ways. First, it may have been created locally from an empty future source. In this case it shares the value object with the future source object and will automatically receive the value when the future source is bound. Secondly, it may have been created as a return value in a proxy method. The ID of this future has then been sent in the method message to the active object. Thirdly, the future may have achieved its pending state when a received future (a method parameter or a part of an unmarshalled data structure or non-active object) is unmarshalled into it. In this case the unmarshalled message contains the ID of the pending future on the other side.

In both latter cases the KC++ run-time system has to keep track of pending futures so that when a future value message is received, the value is unmarshalled into the correct future. Section 5.1.6 explained that the KC++ run-time system maintains a list
of all pending futures to prevent their premature destruction. This list is actually a map data structure linking future IDs to local future objects.

Each time a pending future is created and its value will be received from outside, the run-time system inserts the ID of the future and a future object sharing the value with the created future to the map. When a future value message is received, the ID of the value is extracted from the message and used to find the correct future object in the map. Then the run-time system binds the value to the future and removes the ID and the future from the map.

On the sender side, the need to send a future’s value elsewhere may arise for two reasons. The value may be the return value of a method or the value of a future reference parameter, which must be sent to the caller. Alternatively a pending future (i.e. its ID) may have been sent to another active object, either as an explicit future return value or as a part of a non-active data structure or object.

The value to be sent may become available for two reasons: either a future source is bound to a value, or a local pending future receives its value from outside, and this value must be resent to other active objects. Future sources are internally implemented as futures, so both these cases are actually the same: a local pending future has received its value, and this value must be sent elsewhere.

The KC++ run-time system keeps also a map of all pending futures whose eventual value must be forwarded elsewhere. When a local future receives its value (for whatever reason), the run-time system checks from the map whether the value of the future must be forwarded. The map also contains the destination of the value. The run-time system creates appropriate future value messages, sends them, and then removes the entries from the map, thus forwarding the value to all appropriate active objects.

It should be noted that both the receive and send mechanisms may be triggered at the same time if a pending future has already been sent elsewhere. For example, an active object could receive a pending future as a method return value and then send it to a third active object as a future parameter. When the first method call completes, the future’s value is sent to the caller. There the run-time system first binds the value to the local return value future and then immediately forwards the value to the third active object.

### 6.2.4 Implementation of message streams

The message streams in the KC++ library are responsible for conveying information from one active object to another. The programmer only has to care about message streams if he wants to pass his own non-active types as active object method parameters. In this case he has to write the appropriate marshalling and unmarshalling functions for his type. The KC++ library itself contains all necessary marshalling and unmarshalling functions for all the C++ built-in types, as well as for futures, active objects, and active object pointers and references.
There are two types of message objects: output streams and input streams. An input message stream (IMsg) represents a message received from outside. It contains information about the sender and the type of the message (either a method message or a future value message). The KC++ library uses these to decide how to handle the message, and then uses unmarshalling functions to read the method ID and parameter values (if the message is a method message) or the future value (if the message is a future value message).

Similarly output message streams (OMsg) represent messages to be sent elsewhere. Each message object contains the address of the message's recipient and the type of the message (again either a method or a future value message). Method messages are created by the proxy objects when any of their methods are called. Future value message are created in two places. The active object method invoker creates them to send the return value (or values) when a method has been completed. The KC++ run-time system creates them to forward future values to other active objects. Marshalling functions are used to put necessary parameter values or future values to the message before sending. The output message class has a method send() which takes care of sending the message to its recipient.

It should be noted that even though the names of the classes refer to “messages,” the actual communication do not necessarily have to use a message passing system. If all active objects have a shared memory, it is possible to write the message objects to utilise this and “send” the information using semaphores and shared memory. It is even possible (and planned, see Chapter 8) to write a version of the library where some objects may have shared memory while others do not. The library would then use message passing between two active objects which do not have shared memory, and the more efficient shared memory approach otherwise.

6.2.5 Implementation of unmarshalling

Marshalling into an output stream always happens with an overloaded operator <<. However, there are two possibilities for unmarshalling, as explained in Chapter 4. Normally an unmarshalling constructor should be used, but the programmer can also provide an unmarshalling operator >> and declare it with a special macro called SPECIAL_UNMARSHAL. This section explains the reasons for two unmarshalling mechanisms and describes the implementation of SPECIAL_UNMARSHAL.

Semantically the marshalling/unmarshalling sequence is very close to creating a copy of an object. The only differences are that the copy is possibly created in another address space and that the creation of the actual copy may be delayed (the unmarshalling does not have to happen immediately after marshalling). Copying objects is performed in C++ with copy constructors, so it was considered sensible to make unmarshalling as close to copy construction as possible. The logical choice was to require that unmarshalling is performed in an unmarshalling constructor, which is just like a copy constructor except that it takes an input message object as a parameter.
In many other systems (CORBA, CHAOS++, and UC++, to name but a few) unmarshalling is only possible with an unmarshalling function which unmarshals a received message into *an already existing object*. There are two important semantical differences to copying in this approach. First, unmarshalling into an existing object corresponds to *assignment*, not copying. These two things can be quite different in C++ with some user-defined classes. Secondly, the object to which the message is unmarshalled must be created somehow. Usually the default constructor is used for this purpose, as the part of the library doing the unmarshalling does not know anything about the object to be created.

However, this means that the classes to be unmarshalled must provide default constructors, even if there is no other need for them. It is widely agreed in the C++ community that unnecessary default constructors should be avoided, because they hide some common programming mistakes [Meyers, 1996]. Furthermore, in many cases it may be impossible to write a default constructor which would result in an object which is in a “full working order”. In these cases the programmer has to somehow mark the object as “incomplete” in the default constructor, and then change this state after unmarshalling is complete. For these reasons the unmarshalling constructor was considered a better choice, even though it is not symmetrical with the marshalling function.

The unmarshalling constructor has its limitations, though. It is not possible to write new constructors for built-in types or enumeration types. Similarly it is often impossible to add new constructors to third-party classes. For these cases, the unmarshalling operator `>>` is practically the only choice. However, the code in the KC++ library cannot automatically know whether to use the constructor or the `>>` operator for an arbitrary type.

The problem is solved in KC++ with template specialisation, which makes it possible to treat some types in a different way. The KC++ library code performs unmarshalling using an inline template function called `KCPP_unmarshal`, whose definition is shown in Listing 6.4 on the next page. The function takes two parameters. The first parameter is a pointer to an uninitialised area of memory where the unmarshalled object is to be created (using placement `new`). The second parameter is a reference to the input message object to unmarshal from. The default version of this function simply uses the type’s unmarshalling constructor to unmarshal and create the object.

If the unmarshalling constructor is not an option for a certain type, a template specialisation of `KCPP_unmarshal` can be defined for this type. This specialised function creates the object using its default constructor, and then uses operator `>>` to unmarshal the data to the object. The KC++ library makes this specialisation easy by providing macro `SPECIAL_UNMARSHAL`, which takes the type as a parameter and defines the necessary template specialisation. The library already contains specialisations for all built-in types, and the users can easily declare their own specialisations for types where the unmarshalling constructor is impossible to implement.
template<typename Type>
inline void KCPP_unmarshal(void* position, IMsg& imsg) {
  return new(position) Type(imsg);
}

#define SPECIAL_UNMARSHAL(Type) 
template<>
inline void KCPP_unmarshal<Type>(void* position, IMsg& imsg) 
{
  Type* p(new(position) Type);
  try {
    imsg >> *p;
  }
  catch (...) {
    p->Type(); throw;
  }
}

LISTING 6.4: The definition of SPECIAL_UNMARSHAL
Chapter 7

Related work

This chapter discusses some other work related to KC++. It does not attempt to be a complete survey, but it contains some information that was found during the creation of KC++ and writing of this thesis.

7.1 Classification of concurrent object-oriented systems

There are many different ways to classify and categorise concurrent object-oriented systems. One such classification is presented by Briot, Guerraoui, and Lohr in “Concurrency and Distribution in Object-Oriented Programming” [Briot et al., 1998]. In the article, different approaches to the problem of adding concurrency to object-oriented programming are divided into three categories: the library approach, the integrative approach, and the reflective approach.

In the library approach, concurrency is introduced in the form of a library (usually a class library) which provides necessary services for asynchronous calls, mutual exclusion etc. This approach is briefly discussed in Section 7.3. Although KC++ also contains a library, it does not belong to this category, because concurrency support is added to the program by the KC++ compiler.

The integrative approach adds concurrency by unifying some concurrent concepts with object-oriented ones, thus “mapping” concurrency into object-oriented programming. KC++ is an example of this approach, as the concept of “active objects” is simply an unification of the concepts of “thread of execution” and “object”. The article of Briot et al. mentions that this unification is problematic, because there are easily some conflicts with different concepts. Inheritance anomaly [Matsuoka and Yonezawa, 1993] is mentioned as one example. It means a case where the synchronisation constraints of the base active class must be partially rewritten in a derived active class. This anomaly does not affect KC++, however, because KC++ does not provide synchronisation constraints (the one-thread-run-to-completion semantics makes them unnecessary).
The reflective approach is based on metaobject protocols [Kiczales et al., 1991] and metaprogams. The idea is “to separate the application program from the various aspects of its implementation and computation contexts (models of computation, communication, distribution, etc.).” [Briot et al., 1998]. The reflective approach is probably the most “general” and flexible of these approaches, but it is also quite a new concept, so there are still few concrete systems which would be purely based on this approach.

### 7.2 Concurrent object-oriented languages

Some object-oriented languages have concurrent features built in. This section mentions some common languages and briefly discusses their support for concurrent object-oriented programming.

The Ada language has had support for concurrency from the beginning, and so does the current Ada-95 [Ada, 1995]. The Ada language is not really object-oriented, but object-based like C++. For this reason concurrency support in Ada is not on object level, but the language contains non-object-oriented mechanisms for task creation and synchronisation (“rendez-vous”). These mechanisms are enough to provide support for concurrent programming.

The Smalltalk language has also had basic support for concurrency since Smalltalk-80 [Goldberg and Robson, 1983]. Since Smalltalk is completely object-oriented, primitives for concurrency are also based on classes. A Process class represents actions that can be executed concurrently, and class ProcessorScheduler takes care of actually scheduling the processes. Synchronisation is handled with the class Semaphore, which implements traditional semaphores. The Smalltalk library also provides some other classes based on Semaphore to make concurrent programming easier. However, all method calls are still synchronous, so asynchronous behaviour must be coded explicitly. There are also extensions to Smalltalk concurrency, like ConcurrentSmalltalk [Yokote and Tokoro, 1987].

The Java language [Arnold and Gosling, 1998] has also built in concurrency support. Java is widely used in Internet programming, where reactive programming (in which the program must react to several simultaneous events) is often needed [Lea, 1997]. As a programming language Java is almost completely object-oriented, so it is not surprising that concurrency support is also built using classes and objects.

The class java.lang.Thread is used to initiate and control new activities. Each instance of this class represents a new thread of execution. Keywords synchronized and volatile are used to control how methods are executed concurrently. The Java “base” class java.lang.Object has also methods wait, notify, and notifyAll, which can be used for synchronisation.

Concurrency in Java is based solely on creating new instances of the thread class. As in Smalltalk, all method calls are still synchronous (including the Remote Method Invocation, RMI).
7.3 Concurrent object-oriented libraries

In the library-based approach the underlying object-oriented programming language is not changed at all, but all services related to concurrency are provided in the form of a class library. The library approach is quite useful in very flexible languages like Smalltalk, where several concurrency libraries like Simtalk [Bézivin, 1987] and Actalk [Briot, 1989] have been built on the relative basic concurrency support of the main language.

In less flexible languages like C++ the library approach does not easily provide as “natural” results, because the language has fixed semantics for object-oriented features such as method invocation. However, C++ is such a widely used language that several concurrency libraries have been written for it.

Many such libraries provide a class called “Thread” or “Process”, which can be instantiated to get a concurrently running thread of execution. Usually the programmer is supposed to derive his own classes from this class. The concurrently executed code is written either in the new class’s constructor, like in Sun’s C++ Coroutine Library [Sun, 1996], or in a specialised virtual function.

Active objects are also possible in the library approach. ABC++ [O’Farrell et al., 1996] is implemented as a class library, and has an inheritance-based active object concept like KC++. Active object creation and asynchronous method invocation has to be performed using a special syntax and helper functions, because ABC++ does not touch the source code in any way.

One quite specialised approach is to provide a library for parallel handling of a collection of similar objects, aimed especially for high-performance parallel computing. One such library is the Amelia Vector Template Library (AVTL) [Sheffler, 1996]. AVTL provides containers which resemble the standard C++ containers like vector, and a possibility to perform parallel operations on the elements of containers. AVTL also provides generic function objects which can be used to act on the elements of containers in a parallel manner. The library is designed for distributed-memory environments, so it also takes care of necessary marshalling and unmarshalling between execution locations. The pC++ system [Yang et al., 1996] is also a “collection-based” system like AVTL, but it also provides a small extension to the C++ syntax on top of its class library.

Frameworks are becoming more and more popular in object-oriented programming. It is therefore not surprising that they have also been used to add concurrency to object-oriented programming. POOMA (Parallel Object-Oriented Methods and Applications framework) [Reynders et al., 1996] is such a framework for data-parallel and scientific applications. The programmer inherits his own classes from the framework and uses services provided by it. The framework itself performs necessary data decomposition and communications.

Several existing C-based concurrent libraries have also been ported to C++. These include MPI++ [Skjellum et al., 1996] based on the MPI library, and CHAOS++ [Chang
et al., 1996] based on the CHAOS library. Both add some object-oriented support to the original libraries.

There are many C++ libraries which provide traditional building blocks for concurrency, like semaphores, mutex locks, monitors, remote function calls, etc. Such libraries include the ACE library [Schmidt and Wang, 1995] and Threads.h++ [Rogue Wave, 2000].

Finally, Object Management Group’s CORBA [OMG, 1998a] has also support for concurrency. CORBA is mainly aimed for distributed object-oriented programming, but it also provides mechanisms for asynchronous processing through dynamic invocation interface. The new CORBA Messaging Specification [OMG, 1998b] provides also Asynchronous Method Invocation (AMI), which makes asynchronous method calls easier for the programmer. It contains also future-like mechanisms for delayed return values.

7.4 Concurrent extensions to C++

Just like there are many concurrent object-oriented libraries available for C++, there are at least as many concurrent extensions to the language. Many such extensions are presented in “Parallel Programming Using C++” [Wilson and Lu, 1996]. This section describes the main features of a few such extensions, and briefly compares them with KC++.

The CC++ language [Kesselman, 1996] is a superset of C++. It adds concurrency to C++ with six new keywords. With par, parfor, and spawn programs can create parallel code blocks, parallel loops, and asynchronous function calls. Keywords atomic, sync, and global are used for atomic operations, synchronisation, and shared data.

The new keywords in CC++ are not object-oriented as such, but can of course be used in connection with objects. This gives the programmer a possibility to write concurrent programs in non-object-oriented manner. On the other hand, if active objects, futures etc. are needed, they have to be written by the programmer himself using the concurrency primitives in CC++. An article about CC++ is also found in “Research Directions in Concurrent Object-Oriented Programming” [Chandy and Kesselman, 1993].

The CHARM++ language [Kalé and Krishnan, 1996] is based on the concepts of its predecessor, the CHARM language, which is an extension to C. CHARM++ also implements active objects where only one method may be active at any time. The language adds several new keywords to C++ to mark and create chares (CHARM++’s name for active objects), external interfaces and message passing. All chares are created dynamically, with a syntax similar to C++ new. Data can be shared between chares using several different “shared object types”. CHARM++ also provides support for dynamic load balancing between physical processors, etc.

Unlike the two languages described previously, the COOL language [Chandra et al., 1996] is designed for shared-memory multiprocessor only. This makes the lan-
guage simpler, because communication can be performed through shared memory locations. Concurrency in COOL is implemented with parallel functions, and Hoare monitors and condition variables are used for synchronisation. Parallel functions can also be invoked “serially”, if necessary. Return values from parallel functions are normal return values, not futures, so if synchronisation to the completion of a function or method is needed, it must be expressed explicitly in the program (using condition variables).

Some extensions aimed for high-performance computing concentrate on providing means for operating on elements of a collection in parallel. Section 7.3 already mentioned some class libraries for such purposes. The C++ language [Larus et al., 1996] is a data-parallel extension to C++ with similar features. It extends the C++ syntax with constructs to create matrices etc., and to perform user-defined operations on them in parallel. C++ also provides tools for combining the results from parallel operations and other tools needed in high-performance computing.

Like C++, the Mentat system [Grimshaw et al., 1996] is aimed for high-performance computing. In Mentat, the programmer marks classes with the keyword `mentat` if they require parallel execution. The Mentat run-time system then creates data-dependency graphs, which model dependencies in the program. The run-time system manages execution of objects based on the dependency graphs. Mentat also provides a return-to-future construct which closely resembles futures in KC++, but is restricted to be used for returning values from parallel functions.

The MPC++ language [Ishikawa et al., 1996] takes a somewhat different approach. It extends the syntax of C++ with a few new keywords, which add multi-threading and message-driven execution. However, MPC++ also provides a modification facility called the MPC++ Meta-object Protocol (MOP). This protocol is similar to that described in [Kiczales et al., 1991]. With the protocol programs can introduce new syntactic constructs and redefine existing ones in order to implement new parallel programming constructs.

In many ways, the C++// language [Caromel et al., 1996] is similar to KC++. C++// also uses the existing C++ syntax and uses inheritance to mark classes as active. The C++// compiler also converts C++// code to normal C++ code, and uses wait-by-necessities (a form of future) to synchronise with asynchronous method calls. However, all active objects in C++// have to be created dynamically (requiring special care when exceptions can occur). The wait-by-necessities are not parametrised in C++//, but rather they are created using inheritance. This means that wait-by-necessities cannot be created for built-in types.

Finally, there is the predecessor of KC++, the UC++ language [Winder et al., 1996]. Some features of KC++ come directly from UC++. For example, KC++ active objects use proxy objects and the same one-thread-run-to-completion semantics as UC++ active objects. However, UC++ does not use type system to mark objects as active. Any object can be made active in UC++ by creating it dynamically with a new operator `activexnew`. Again, this means the programmer has to be careful with exceptions. Nor-
mal C++ method calls are always synchronous in UC++. If asynchronous method call is needed, the language provides an alternative syntax to achieve this. Future types are not used, so asynchronous calls do not have return values. UC++ proxy classes are implemented as special cases of the active classes themselves, making them heavyweight, as they contain all data members of the active classes themselves.
Chapter 8

Conclusions and future work

This thesis has presented KC++, a concurrent programming system based on the C++ language and using active objects for concurrency. It adds concurrency to C++ without extending the language syntax. The KC++ system is not restricted to environments with shared memory, but each active object may have its own private address space. This includes distributed computing where active objects resides in different machines and communicate with each other using a computer network. Future types are used extensively in KC++ to provide asynchronous method calls and synchronisation between active objects.

8.1 Advantages of KC++

The KC++ system in its current state has shown that it is possible to add active objects and concurrency to the C++ language in a way that feels “natural,” i.e. in a way that does not unnecessarily restrict the use of other language features or commonly used C++ programming idioms. This “non-restrictiveness” is quite important in a language like C++, because one of the design goals of the language itself has been been to make different language features non-conflicting and “orthogonal”, leaving it up to the programmer to decide how to use the features [Stroustrup, 1994, p. 113].

8.1.1 Familiar syntax and “feel”

The familiar syntax makes it easier to start using KC++, as the programmer does not have to learn any new peculiarities added to the already complex C++ syntax. Of course KC++ programmers still have to have good knowledge on concurrent programming, because KC++ does not give complete protection from usual concurrent programming mistakes such as dead-locks.

Active objects in KC++ can be handled exactly the same way as normal objects. They can be created dynamically or statically, and pointers and references to active
objects behave in the usual way. This is important because many C++ programming idioms and practises often rely on the way objects are created, passed as parameters etc. These include design patterns, exception-safe programming, generic programming, and many others.

8.1.2 Making old classes active

KCC++ makes it possible to take an existing class and derive an active class from it, allowing programmers to use already existing class libraries concurrently. This feature of KCC++ should be used with utmost care, however, as the interface of many non-concurrent classes simply isn't suitable for concurrent use.

8.1.3 Ease of debugging

The C++ code produced by the KCC++ compiler is almost identical to the original source code. This is a benefit when KCC++ programs are tested and debugged. Standard C++ debuggers can be used to debug KCC++ programs interactively and step through the program line by line. The only restriction is that most debuggers are probably only able to debug one execution thread at a time (although there are debuggers which can handle several concurrent threads at once). When a problem is found and located, the #line directives and the similarity between the produced C++ code and the original KCC++ program makes it easy to find the location in the original program. The produced C++ code is human-readable, so it can also be used as a base of debugging, if the programmer wants to step through and inspect the produced code instead of the original KCC++ source.

8.1.4 Platform independence

The code produced by the KCC++ compiler is standard C++ without any platform-specific extensions or dependencies. All such platform-specific features are implemented in the KCC++ library. This makes it quite easy to port KCC++ to other platforms. It also means that if multiple platforms are needed, the KCC++ program can be compiled only once, and the resulting C++ code can then be compiled for each platform using a native C++ compiler, and linked with the KCC++ library for that platform.

As much of the KCC++ library as possible has been made platform-independent. Proxies and futures are implemented using a simple stream-based communications subsystem, which can be implemented to use message passing (using either connectionless or connection-based communications), shared memory, or other mechanisms. If several such subsystems are implemented, changing the communications mechanism only requires the programmer to link the KCC++ program with the chosen subsystem.
CHAPTER 8. CONCLUSIONS AND FUTURE WORK

8.1.5 Only one communication and synchronisation mechanism

Futures are used throughout the KC++ system for delayed value passing and synchronisation. In addition to asynchronous method calls, programmers can use future sources to use futures explicitly for more complex synchronisation techniques. Using a single synchronisation mechanism makes KC++ simpler and easier to use.

8.2 Disadvantages of KC++

Like any other system (including C++ itself), KC++ has its own disadvantages. Some of them are dictated by the C++ language, some are conscious design decisions, and some are simply things which are not yet fixed in the current version of KC++. This section mentions some of those disadvantages and gives a short analysis of each of them. Some of the disadvantages are planned to be fixed in future versions of KC++, in which case the future plans are not discussed in this section, but are saved to Section 8.3.

8.2.1 The active object model and RTC

Maybe the most important “disadvantage” comes from the active object model used in KC++. As mentioned in Chapter 1, there are several ways to add concurrency to an object-oriented programming language, and different people have different opinions on the best way. The one-method-run-to-completion semantics (RTC) used in KC++ is perhaps the most restrictive way to implement active objects. Some people may consider it even too restrictive, because an active object method cannot “pause” and give way to other methods, but its code must be run to completion before other methods can be executed. In some cases it would also be useful to have several concurrently running methods in an active object (for example in a buffer where only one writer but several readers are allowed).

There are two main reasons why RTC was chosen. It was already used in UC++, which was the “mother” of KC++. Therefore the whole issue was not even considered during the early stages. RTC is also the easiest and most efficient to implement. When each active object consists of a single non-interruptible thread of execution, each active object can simply be implemented as an operating system process or thread. There are also no mutual exclusion problems with the internal data of the active object. The one-to-one relationship between an active object and an execution thread also forces the programmer to explicitly decide how many concurrent processes the program has (or to write code which makes this decision). This follows the C++ ideology where “the programmer is allowed to decide.” [Stroustrup, 1994]
8.2.2 Active object reference counting

One real problem in current KC++ is its way to determine lifetimes of active objects. The current reference counting system means that an active object remains “alive” until there are no proxies referring to it in the whole system. Active object pointers and references also count as proxies, so an active object is not destroyed if even one pointer or reference pointing to it remains. Like in all systems based on reference counting (current Python [van Rossum, 1999], for example), this creates problems with circular references. If two active objects refer to each other, they are not destroyed even if the rest of the system does not contain proxies to them. One solution to this would be to write a “reset” method to one of the objects. This method would remove the reference to the other object. The program would then have to call this method explicitly when the objects are no longer needed.

The reason for reference counting in KC++ is again simplicity and efficiency. Reference counts are easy to keep up to date and do not require a distributed garbage-collection system to decide when an active object may be destroyed. Current distributed garbage-collectors are quite complex and non-efficient, so they would probably not be suitable for a C++-based system.

Another possibility would have been to use the normal C++ semantics and destroy an active object when the “primary” proxy (the proxy which originally created the object) is destroyed. This approach also has its problems because of concurrency. As discussed in Section 4.3.1, it is possible to create an active object, pass a reference to it to another active object as a method parameter, and then destroy the object. The method call is asynchronous, so it is possible (and even probable) that the reference parameter would still be in use when the primary proxy destroys the active object. This approach is used in CC++ [Kesselman, 1996], which also suffers from the problem described above.

8.2.3 No subtyping with non-active bases

When an active class is derived only from the base class Active or other active classes, normal C++ subtyping rules are respected, and an instance of the derived active class is also an instance of the base active class (the “is-a” relationship). Unfortunately this relationship cannot be allowed between a normal non-active base class and an active class derived from it. The reasons for this restriction were discussed in detail in Section 3.3, but the main point is that a derived active object is not an instance of a non-active base class because of asynchronous method calls.

The lack of “non-active” subtyping is sometimes unfortunate if the programmer wants to “drop in” active classes into an existing program by deriving these active classes from existing non-active classes, and then use these classes through base class pointers or references. However, it is almost always impossible to simply add concur-
rency into an existing program without restructuring and reanalysing the existing code, so the restriction in KC++ is not likely to cause problems.

### 8.2.4 The future relaying mechanism

KC++ allows passing pending futures from an active object to another. When the value of a future becomes available, the active object forwards the value of the future to all other active objects to which it had passed the original future. These active objects may in turn again forward the value. If this chain of active objects gets very long, value forwarding may cause considerable traffic between active objects, especially if the value is a large non-active object or a data structure.

In the worst possible case only the last active object in the chain actually needs the value, and all the other active objects just pass on the value without storing it. In this case the value will nevertheless pass through the whole chain of active objects on its way to the final destination.

It would be possible to coordinate the future value passing better, so that the creator of the pending future would always be informed when another active object becomes interested in the value of the future, and when an active object loses interest in it. This "notification mechanism" would be useful with long future chains, but it too would add to the traffic between active objects. With short chains and simple future values the notification mechanism could actually increase communications instead of decreasing it. It was therefore considered that the simple forwarding system used in KC++ was enough.

### 8.3 Future work

Like any piece of software, KC++ is probably never "complete and perfect." This section mentions some improvements and changes which are planned for the future versions of KC++.

It has already been mentioned that the reference counting system, which is used to determine the lifetime of active objects, causes problems with circular reference chains. One possible solution would be to follow normal C++ semantics and let the active object be destroyed when the proxy which created it gets destroyed, even if there are still pointers or references pointing to it elsewhere. Specialisations of C++ `auto_ptr` template could then be used to allow the "ownership" of the active object to be passed from one place to another. Similarly specialised smart pointer classes could be added to allow "shared ownership" using a reference counting solution similar to the current one.

The current implementation of KC++, and especially its run-time library, is only meant for testing purposes. In order to test the performance of the KC++ system, a "quality" implementation of the KC++ library is obviously needed. The implementa-
tion should also be optimised as much as possible so that performance measurements could be carried out.

Exceptions are not currently propagated between active objects. However, exceptions are rapidly becoming a standard way of reporting errors in C++, so it would be important to allow active object methods to report error situations to the caller using exceptions. It is not easy to add exceptions to KC++, because method calls are asynchronous. This means that the caller has already continued executing its code (and maybe already left some try-blocks), making it difficult to decide where in the code the exception should propagate.

One possible future direction would also be to abandon the strict RTC semantics and allow active object methods to pause their execution and allow another method to be executed until a certain synchronisation condition is fulfilled (i.e. probably a future gets its value). This would mean that a single active object could have several non-completed methods, and therefore several threads of execution. However, this change requires probably more work than others mentioned in this section.
Bibliography


[Gamma et al., 1996] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison Wesley, Reading, 1996.


BIBLIOGRAPHY


Appendix A

Producer–consumer buffer from non-active base

This appendix provides an example, which shows how active classes can be created from existing non-active classes using multiple inheritance. The example also demonstrates why this usually creates problems, caused by the fact that non-active classes are not designed with concurrency in mind.

Below is a definition of class Buffer, which is a simple class which allows users to put integers into the buffer and later read them (in FIFO order). The code is from file oldbuffer.h.

```cpp
#include <list>

// A simple buffer
class Buffer
{
public:
  Buffer();
  ~Buffer();

  void put(int value); // Writers use this

  bool isEmpty() const; // true if the buffer is empty
  int get(); // Readers use this

private:
  list<int> values;
};
```

A new active class can be created from Buffer by deriving it from both Buffer and the KC++ class Active. Two other active classes are defined. Producer objects write
number from 1 to 10 into the buffer given in the constructor. Consumer objects in turn read integers from the buffer until the buffer becomes empty. Because both classes are active, producers and consumers can operate in parallel. This code is from file `actbuffer.h`.

```cpp
#include "kcpp_active.h"
#include "oldbuffer.h"

// An active buffer created from the non-active one
class ActBuffer : public Buffer, public Active {
    // No changes necessary, inherits everything from Buffer
};

class Producer : public Active {
public:
    Producer(ActBuffer& buffer);
    ~Producer();
    void produce(); // This function does the actual producing

private:
    ActBuffer& buffer_;
};

class Consumer : public Active {
public:
    Consumer(ActBuffer& buffer); // Consumer gets the buffer it reads from
    void consume(); // This reads values until buffer is empty

private:
    ActBuffer& buffer_;
};
```

Finally, the next code defines the methods in Producer and Consumer. The actual buffer class `ActBuffer` does not need any code, because it inherits everything from `Buffer`.

The `Producer::produce` code is quite simple. It simply calls `put` ten times. Because the buffer is an active object, all method calls are asynchronous and the producer writes the numbers without waiting for previous put operations to complete.
The code in Consumer::consume is more complex. Data can be read from the original Buffer class by first checking that data exists (isEmpty) and then reading it (get). However, ActBuffer is an active object, so it is possible that there are several consumers working in parallel. If the normal non-parallel algorithm would be used, one consumer could check that the buffer is not empty, then another consumer could read the value, emptying the buffer. After this the first consumer would attempt to read a value from a now empty buffer, causing an error condition. This problem can be solved by locking the buffer object for the duration of the isEmpty-get pair.

This code also reveals that the original buffer class was not really suited for parallel use. If both producers and consumers work in parallel, it is possible that the buffer becomes temporarily empty, even if producers still exist. The ActBuffer class has no way of noticing this, so it cannot inform the consumers that more values will become available. Appendix C contains another buffer example which is designed for parallel use.

```cpp
#include "kcpp_activelock.h"

#include "actbuffer.h"

// No code needed for the ActBuffer class

// The Producer class
Producer::Producer(ActBuffer& buffer) : buffer_(buffer) {
}

Producer::~Producer() {
}

void Producer::produce()
{
    for (int i=0; i<10; ++i)
    {
        buffer_.put(i);
    }
}

// The Consumer class
Consumer::Consumer(ActBuffer& buffer) : buffer_(buffer) {
}
```
void Consumer::consume()
{
  // Reads values until the buffer is empty
  while (true)
  {
    // The buffer must be locked so that nobody “steals” the value
    ActiveLock lock(buffer_); // (Section 5.6)

    if (!buffer_.isEmpty()) { break; } // Leave if buffer empty

    int value = buffer_.get(); // Synchronisation
    // Here value could be used
  }
}
Appendix B

Priority semaphore as an active object

This appendix presents a “priority semaphore” synchronisation mechanism implemented as an active object. It shows how future sources can be used and how C++ containers can be used with futures. The `P` operation in the semaphore returns a `void-future`. The caller of `P` is expected to wait using the return value future (i.e. `sem.P().wait()`). Alternatively the caller may of course poll the future occasionally. The `V` operation then binds this future source and releases the waiter. The `P` operation has a priority as its parameter, and waiters are released in priority order.

The semaphore contains a priority queue — a “wait queue” which contains future source-priority pairs. It also contains a count which counts “free” resources (i.e. the normal counter used in non-binary semaphores). The `P` operation first checks whether there are free resources. If the counter is above zero (free resources exist), the operation simply returns an already bound future, so that the caller is released immediately. If the counter is zero, the operation adds another future source-priority pair to the queue, and returns a `void-future` created from the future source.

Similarly the `V` operation first checks whether the wait queue is empty. If it is, the operation simply increments the counter. Otherwise it takes the first future source (with the highest priority) from the queue and binds it, releasing the waiter.
```cpp
#include <deque>
#include <queue>
#include <utility> // For pair and make_pair
#include "kcpp_active.h"
#include "kcpp_future.h"

class Semaphore : public Active
{
public:
    Semaphore(unsigned int newcount);
    // Callers of P may give a waiting priority
    Future<void> P(int priority = 0);
    void V();

private:
    // The type of queue items (priority-future source pairs)
    typedef pair<int, SFuture<void> > QueueItem;
    // A function object for comparing queue items
    struct QueueCompare
    {
        bool operator()(const QueueItem& q1, const QueueItem& q2);
    };
    unsigned int count;
    priority_queue<QueueItem, deque<QueueItem>, QueueCompare> waitqueue;
};

bool Semaphore::QueueCompare::operator()(const QueueItem& q1, const QueueItem& q2)
{
    return q1.first < q2.first; // Compare only priorities
}

Semaphore::Semaphore(unsigned int newcount)
    : count(newcount), waitqueue()
{
}

Future<void> Semaphore::P(int priority)
```
APPENDIX B. PRIORITY SEMAPHORE AS AN ACTIVE OBJECT

```cpp
{ if (count > 0) 
  { // No waiting necessary 
    --count;
    return Future<void>(); // Return a non-pending future
  }
else 
  { // Waiting necessary, add to queue 
    SFuture<void> waitfut;
    waitqueue.push(make_pair(priority, waitfut));
    return waitfut; // Return a pending future
  }
}

void Semaphore::V()
{
  if (waitqueue.empty()) 
    { // Nobody waiting, increment count 
      ++count;
    }
else 
  { // Otherwise signal the first waiter 
    SFuture<void> waitfut = waitqueue.top().second;
    waitqueue.pop(); // Remove the future source from queue 
    waitfut.bind();
  }
}```
Appendix C

Producer–consumer buffer with futures

This appendix presents a future-based concurrent buffer. It shows how futures can be used to provide maximal concurrency. It also solves the problems of the buffer in Appendix A.

Each producer registers itself to the buffer before it begins and unregisters when it stops. The buffer maintains a count of registered producers. The buffer considers itself “finished” when the buffer is empty, all producers have unregistered, and there has been at least one producer (so that the buffer is not “finished” right from the start). The buffer also keeps a list of values stored in the buffer. These values are integer futures, so the producers can promise to provide values into the buffer. The buffer then simply delivers these “integer promises” to the consumers, letting the KC++ runtime system to do all the dirty work. The description of put and get operations below show that futures are also used to let consumers get values the buffer does not even know about yet.

Each consumer reads data using the get method. The method returns a future of a type ValuePair. This pair tells (when its value becomes available) whether the buffer is finished (field isFinished) and if not, a future which will contain the value read from the buffer (field value). When get is executed, it first checks whether the value list is currently empty, and if not, it simply returns the first value in the list (a future, actually). If the list is empty, the operation checks whether there are any registered producers left. If there are none, the buffer is finished and the consumers is informed about this. However, if the value is empty and there still are producers, it is possible that more values will become available. For this reason the buffer contains a “wait list” waiting. In this case a new future source of type ValuePair is created and added to the list, and the return value of the operation is created from this future source.
APPENDIX C. PRODUCER–CONSUMER BUFFER WITH FUTURES

When a put operation is executed, it first checks whether the wait list is empty. If it is, the value (a future) is simply inserted to the value list. If the wait list contains waiters, the operation binds the first future source in the wait list to the value (a future) written to the buffer, causing the value to be sent to the caller of the get operation.

Finally, it is possible that there are still waiters in the wait list when the last producer unregisters. In this case all waiters are released and informed about the status of the buffer using the private method `releaseWaiters`.

```cpp
#include <list>
#include <cassert>
#include "kcpp_active.h"
#include "kcpp_future.h"
#include "kcpp_messages.h"

// Struct which is the return value of buffer’s get operation
struct ValuePair
{
    bool isFinished;  // All producers have finished and buffer is empty?
    Future<int> value; // Eventual read value, if buffer not finished

    // Constructors to make creating ValuePairs easier
    ValuePair() : isFinished(true), value() {}
    ValuePair(bool f, Future<int> v) : isFinished(f), value(v) {}

    // Unmarshalling constructor (Section 4.1)
    inline ValuePair(IMsg& imsg);
};

inline ValuePair::ValuePair(IMsg& imsg) : isFinished(false), value(imsg)
{
    imsg >> isFinished;  // Must use >> for built-in types
}

// Marshalling operator
inline OMsg& operator <<(OMsg& omsg, const ValuePair& p)
{
    omsg << p.value << p.isFinished; // Order matches the unmarshalling
    return omsg;
}

// Unmarshalling operator (not actually needed by KC++)
inline IMsg& operator>>(IMsg& imsg, ValuePair& p)
```
```cpp
    {  
      imsg >> p.value >> p.isFinished;  
      return imsg;  
    }

  
  class Buffer : public Active
  {  
    public:  
      Buffer();  
      ~Buffer();  

      // Producers use these to register and unregister to/from buffer  
      void registerProducer();  
      void unregisterProducer();  

      void put(Future<int> value); // Producers use this  
      Future<ValuePair> get(); // Consumers use this  

    private:  
      void releaseWaiters(); // Releases waiters when all producers have gone  

      bool justStarted_; // true after creation but before first registration  

      unsigned int producerCount_; // How many producers registered  

      list<Future<int>> values_; // Place to put written values  
      list<SFuture<ValuePair>> waiting_; // Wait queue for consumers  
  }

  // A dummy active class which (somehow) generates values for the buffer  
  class Generator : public Active
  {  
    public:  
      int generate(int i);  

    private:  
  }

  class Producer : public Active
  {  
    public:  
      Producer(Buffer& buffer, Generator& gen);  
      ~Producer();  

      void produce(); // This function does the actual producing  

    private:  
  
```
class Consumer : public Active {
  public:
    Consumer(Buffer& buffer); // Consumer gets the buffer it reads from
    void consume(); // This reads values until buffer finishes
  private:
    Buffer& buffer_;
};

// The Buffer class
Buffer::Buffer() : justStarted_(true), producerCount_(0), values_(), waiting_()
{
}

Buffer::~Buffer()
{
  releaseWaiters(); // Just release any waiting consumers (if any)
}

void Buffer::registerProducer()
{
  assert(justStarted_ || producerCount_ > 0);
  justStarted_ = false; // At least one producer registered
  ++producerCount_;
}

void Buffer::unregisterProducer()
{
  assert(producerCount_ > 0);
  --producerCount_;
  if (producerCount_ == 0) { releaseWaiters(); } // No more producers
}

void Buffer::put(Future<int> value)
{
  if (waiting_.empty())
APPENDIX C. PRODUCER–CONSUMER BUFFER WITH FUTURES

```cpp
// No consumers waiting, put into the value list
values_.push_back(value);
} else {
  // A consumer is waiting, send the value to it
  waiting_.front().bind(false, value);
  waiting_.pop_front(); // Remove consumer from wait queue
}
}

FutureValuePair Buffer::get()
{
if (!values_.empty())
  // Values in the list, return the first
  Future<int> value(values_.front());
  values_.pop_front();
  return ValuePair(false, value); // Return future get its value here
} else if (!justStarted_ && producerCount_ == 0)
  // No producers either, we are finished
  return ValuePair(true, Future<int>());
else
  // Otherwise enter the wait queue
  SFutureValuePair waitSource; // New future source for waiting
  waiting_.push_back(waitSource); // Add it to the wait queue
  return waitSource; // Create return value from the future source
}

void Buffer::releaseWaiters()
{
  // Inform all waiters that values have run out
  while (!waiting_.empty())
  {
    waiting_.front().bind(true, Future<int>()); // Release waiter
    waiting_.pop_front();
  }
}

// The Producer class
Producer::Producer(Buffer& buffer, Generator& gen)
  : buffer_(buffer), gen_(gen)
  { buffer_.registerProducer();
```
APPENDIX C. PRODUCER–CONSUMER BUFFER WITH FUTURES

```cpp
// Producer::~Producer()
Producer::~Producer()
{
buffer_.unregisterProducer();
}

void Producer::produce()
{
    for (int i = 0; i < 10; ++i)
    {
        buffer_.put(gen_.generate(i)); // generator returns a future
    }
}

// The Consumer
Consumer::Consumer(Buffer& buffer) : buffer_(buffer)
{
}

void Consumer::consume()
{
    // Reads values until all producers have unregistered
    while (true)
    {
        Future<ValuePair> result;
        result = buffer_.get(); // Read (asynchronous)
        // Here might be some other processing
        ValuePair valuepair(result); // Synchronisation
        if (valuepair.isFinished) { break; } // Stop if finished
        // Here we know that the value will become available
        // Some processing again
        int value = valuepair.value; // Synchronisation with the producer’s value
        // Here value could be used
    }
}
```