Building object-oriented software with the D-Bus messaging system

UNIVERSITY OF TURKU
Department of Information Technology
Computer Science
Master of Science Thesis
June 18, 2012
Olli Salli

UNIVERSITY OF TURKU
Department of Information Technology

SALLI, OLLI: Building object-oriented software with the D-Bus messaging system

Master's Thesis, 100 p.
Computer Science
June 2012

Object-oriented programming is a widely adopted paradigm for desktop software development. This paradigm partitions software into separate entities, *objects*, which consist of data and related procedures used to modify and inspect it. The paradigm has evolved during the last few decades to emphasize decoupling between object implementations, via means such as explicit interface inheritance and event-based implicit invocation.

Inter-process communication (IPC) technologies allow applications to interact with each other. This enables making software distributed across multiple processes, resulting in a modular architecture with benefits in resource sharing, robustness, code reuse and security. The support for object-oriented programming concepts varies between IPC systems. This thesis is focused on the D-Bus system, which has recently gained a lot of users, but is still scantily researched. D-Bus has support for asynchronous remote procedure calls with return values and a content-based publish/subscribe event delivery mechanism.

In this thesis, several patterns for method invocation in D-Bus and similar systems are compared. The patterns that simulate synchronous local calls are shown to be dangerous. Later, we present a state-caching proxy construct, which avoids the complexity of properly asynchronous calls for object inspection. The proxy and certain supplementary constructs are presented conceptually as generic object-oriented *design patterns*. The effect of these patterns on non-functional qualities of software, such as complexity, performance and power consumption, is reasoned about based on the properties of the D-Bus system. The use of the patterns reduces complexity, but maintains the other qualities at a good level.

Finally, we present currently existing means of specifying D-Bus object interfaces for the purposes of code and documentation generation. The interface description language used by the Telepathy modular IM/VoIP framework is found to be an useful extension of the basic D-Bus introspection format.

Keywords: D-Bus, proxy object, design patterns, interface description language, inter-process communication, object-oriented, Telepathy

TURUN YLIOPISTO
Informaatioteknologian laitos

SALLI, OLLI: Oliopohjaisten ohjelmistojen rakentaminen D-Bus-viestintäjärjestelmää käyttäen

Pro gradu -tutkielma, 100 s.
Tietojenkäsittelytiede
Kesäkuu 2012

Oliopohjainen ohjelmointi on laajasti käytetty menetelmä työpöytäohjelmistojen toteuttamiseen. Tässä menetelmässä ohjelmisto jakautuu erillisiin osiin, *olioihin*, jotka koostuvat tiedosta ja siihen liittyvästä toiminnallisuudesta, jota käytetään tiedon muokkaamiseen ja tarkasteluun. Menetelmä on kehittynyt viimeisten vuosikymmenten aikana painottamaan olioiden toteutusten erottamista toisistaan, tavoilla kuten rajapinnoista periminen ja tapahtumapohjainen implisiittinen kutsuminen.

Prosessienvälisen kommunikaation (IPC) menetelmät mahdollistavat sovellusten kommunikoinnin toistensa kanssa. Tämän ansiosta ohjelmistot voidaan hajauttaa useaan prosessiin, jolloin syntyy modulaarinen arkkitehtuuri, mistä on hyötyä resurssien jakamiselle, selviytymiskyvylle, toteutuksen uudelleenkäytölle ja turvallisuudelle. Tuki oliopohjaisen ohjelmoinnin periaatteille vaihtelee IPC-järjestelmien välillä. Tässä opinnäytteessä keskitytään D-Bus-järjestelmään, joka on viime aikoina saanut paljon käyttäjiä, mutta on yhä vahäisesti tutkittu. D-Bus-järjestelmässä on tuki asynkronisille etäproseduurikutsuille paluuarvoin ja sisältöpohjainen julkaisija/tilaaja-tyyppinen tapahtumienvälitysmekanismi.

Opinnäytteessä vertaillaan muutamia malleja metodien kutsumiseen D-Busissa ja samankaltaisissa järjestelmissä. Synkronisia paikallisia kutsuja matkivien mallien näytetään olevan vaarallisia. Myöhemmin esitellään tilaa peilaava *edustajaoliorakenne*, joka kiertää aitojen asynkronisten kutsujen monimutkaisuuden olioiden tarkastelussa. Edustajaolio ja eräät täydentävät rakenteet esitetään käsitteellisesti yleisinä oliopohjaisina *suunnittelumalleina*. Näiden mallien vaikutusta ohjelmistojen ei-toiminnallisiin ominaisuuksiin, kuten monimutkaisuuteen, suorituskykyyn ja virrankulutukseen, arvioidaan D-Bus-järjestelmän ominaisuuksiin perustuen. Mallien käyttäminen vähentää monimutkaisuutta, mutta säilyttää muut ominaisuudet hyvällä tasolla.

Lopuksi tarkastellaan olemassa olevia menetelmiä D-Bus-oliorajapintojen määrittelyyn ohjelmakoodin ja dokumentaation generointia varten. Modulaarisen Telepathy-viestintäohjelmakehyksen käyttämä *rajapintojen määrittelykieli* havaitaan hyödylliseksi laajennokseksi D-Bus-introspektiomuotoon nähden.

Asiasanat: D-Bus, edustajaolio, suunnittelumallit, rajapintojen määrittelykieli, prosessienvälinen kommunikointi, oliopohjaisuus, Telepathy

Contents

List of Figures

List of Tables

List of Symbols

APM	Arguments per member	63
C_c	Constraint complexity	62
C_g	Configuration complexity	62
C_s	Signature complexity	62
IC	Interface complexity	62
n_e	Number of events in the interface	63
n_o	Number of operations in the interface	63
CPU	Central processing unit	29
GUI	Graphical user interface	55
IDL	Interface description language	35
IM	Instant messaging	2
IPC	Inter-process communication	1
ISP	Interface segregation principle	21
NAT	Network address translation	26
RPC	Remote procedure call	26
UI	User interface	76
UML2	Unified Modeling Language version 2	60

1 INTRODUCTION

Computer desktop environments have traditionally been seen as collections of applications that are implemented using a common infrastructure. However, the applications are just means to an end—they are utilized by users to achieve the goals in their personal *use cases* for the system. A single application can seldom offer all the functionality required for a particular use case without becoming excessively complicated to use for others. Furthermore, large *monolithic* applications end up duplicating a lot of their implementation with other applications with overlapping functionality. Thus, it is beneficial to combine smaller software components together, so that they can be used to perform a real-world task.

Software libraries allow factoring out common parts of application implementations to modules that are shared between multiple applications. Libraries consist of constructs such as methods, *object-oriented* classes, and data. For example, the visual representation code of the lower-level graphical user interface components such as buttons and forms can be shared by the applications in a desktop environment. Such reusable elements are typically implemented as object-oriented classes. In general, object-oriented programming has been widely adopted in desktop application development due to the possibilities it offers for creating intuitive analogues of real-world entities.

Inter-process communication (IPC) technologies, such as CORBA, DCOP, and D-Bus, allow applications to interact with each other. Such interaction allows *delegating* parts of the tasks that an application is required to perform to other applications, which act as *services*. Some desktop service applications might be stand-alone *backends*, with no directly accessible user interface. Multiple user-visible *frontend* processes can access a single backend concurrently.

Both utilizing software libraries and delegating duties to backends via IPC help avoid duplication of implementation effort in applications. However, while applications employing a library's services end up duplicating its runtime state, a single backend's state and the results it produces can be consumed by any number of frontend consumers simultaneously, conserving *resources*. This is especially important in processing power, working memory and electrical energy constrained environments, such as those present in the emerging smart phone and mobile internet device fields.

Some resources cannot be concurrently accessed by multiple applications. A backend can be used to expose such a resource as a service, governing access to it. As a practical example, many online real-time communications services, such as Windows Live Messenger, only allow one connection to be made to a user account at a time. Unless this connection is shared via a backend process that serves as an intermediary, only a single application can communicate through the online service simultaneously. Such online services however offer functionality related to multiple different use cases, such as voice and video calling, file transfer and sharing media such as images, in addition to textual conversations. Incorporating all that functionality to a single application, as would be necessary unless a frontend-backend model is utilized, would require a very large and complicated user interface and would be a significant implementation and maintenance burden.

Thus, refactoring traditional monolithic applications to a multitude of frontend and backend processes, connected using IPC, can enable a more task-oriented division of desktop functionality. Continuing the instant messaging example, a minimal taskbar applet application could be constantly running to show and allow changing the user's online status in the network. Another purpose-built application could be used to view the list of contacts reachable through the account, and a yet separate one would launch to handle a call placed on one of them. All three applications could share a single network connection to the messaging network, maintained by a backend. The Telepathy framework [dTdCDG+12] is a practical generalized implementation of online real-time communication functionality, such as instant messaging (IM), built around the D-Bus IPC system.

While IPC enables new kinds of application interaction patterns, its use also gives rise to many challenges that are not present in the design of monolithic applications. Invoking remote methods and especially transferring the results back carries a significant *performance overhead* compared to calling code in the local process. *Concurrent programming* pitfalls, such as the possibility for deadlocks and race conditions, also apply because the frontend and backend processes run independently. Overcoming these drawbacks can increase programming complexity.

Whereas a service backend encompassing a given set of functionality is usually implemented just once, the challenges in accessing it via IPC are present in all of its frontends. Suitable IPC protocol and service access interface design enables providing service-specific *client libraries* that ease this burden and hence further the use of desktop services. A client library is a software library specifically created to offer a more natural interface to access backend services. Ideally, they could make remote objects as simple to use programmatically as local ones, while handling the IPC communication behind the scenes in a correct and efficient fashion.

1.1 Problem Definition

There are multiple existing distributed software frameworks, built around various kinds of IPC systems to fulfill desktop use cases. This report attempts to define the most advantageous IPC communication patterns for building such systems. The three mutually conflicting viewpoints of performance/scalability, correctness and programming convenience are considered.

The D-Bus IPC system has gained extensive practical adoption during the last few years, especially on Linux-based platforms. However, there is little academic research available on the subject of this system. We will study the properties of D-Bus in the context of existing research on other IPC technologies.

As the widespread adoption of desktop services is more dependent on frontend than backend implementation convenience, a specific focus is given to design techniques for the client libraries used by frontends to access the services. The goal is to identify and formalize object-oriented *design patterns* specifically related to inter-process object access. The design of the components of the Telepathy framework is used here as a source of ideas and examples.

Additionally, we explore methods to automate some parts of the implementation of client libraries. This is centered around a study of languages for specifying D-Bus interfaces.

These focus areas of the research can be summarized as the following questions:

- How do IPC systems in general and D-Bus in particular support object-oriented programming?
- How should IPC messaging be designed to ensure reasonable performance and reliability?
- Which kinds of design patterns are useful for building well-behaving object-oriented software with the D-Bus system?
- How can the implementation effort of D-Bus communication be kept at a manageable level?

1.2 Project Organization

This report has been authored by Olli Salli, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the University of Turku. The research leverages work done for and is supported by Collabora Limited, a software consultancy specializing in open source technologies and principles.

The work is supervised at the Department of Information Technology in the University of Turku by Professor, Ph.D. Olli Nevalainen. This final report has additionally been inspected by Professor, Ph.D. Ville Leppänen.

1.3 Report Structure

Chapter 2

In Chapter 2 we present, through literary review, core object-oriented programming concepts, as relevant to our study of object-oriented systems in a distributed setting. We also make observations on certain challenges in the practical implementation of these concepts. These will be very relevant in the inter-process context.

Chapter 3

In this chapter, inter-process communication is explored. The support for generic object-oriented concepts in existing IPC systems is evaluated. There is a specific focus on the properties of the D-Bus message bus system.

Chapter 4

Chapter 4 introduces the formal methods we will later use for describing implementation patterns of object-oriented distributed software, and for evaluating their impact on software complexity.

Chapter 5

In this chapter, we propose some architectural guidelines for the design of D-Bus client software. This is be centered around an extension to the proxy pattern of Shapiro [Sha86] for structuring distributed systems. We study implementing the extended proxy pattern using D-Bus facilities, and formally evaluate its impact on non-functional qualities of resulting systems. Also, some other constructs directly related to the proxies are explored, as are implications on D-Bus interface design.

Chapter 6

In this chapter, we describe some methods to describe D-Bus interfaces in a machine readable fashion. We present a sophisticated interface description language (IDL), from which documentation and lower-level D-Bus client and service code can be automatically generated.

Chapter 7

Here, we review key points of the research and summarize the findings.

2 OBJECT-ORIENTED PROGRAMMING

The object-oriented paradigm of programming partitions software into separate runtime entities, *objects*, which consist of data and related procedures used to inspect and modify it [Coh84]. Objects belong to one or more *classes*, which serve as templates for object *instances* (Section 2.1). The ability to extend classes via *inheritance* (Section 2.2) completes the classical definition of an object-oriented language [Weg87]:

object-oriented = objects + classes + inheritance

Delegation (Section 2.3) can be thought of as a more dynamic form of inheritance. Inheriting from *explicit interfaces* (Section 2.4) and having *events* (Section 2.7) as object interface members are more recent refinements to the object-oriented paradigm. All three of these concepts attempt to decrease coupling between object implementations. This decoupling improves modularity and code reusability, as will be discussed later. Finally, the *design by contract* philosophy (Section 2.5) and *exception handling* (Section 2.6) deal with corner-cases during the execution of object-oriented software.

Object-oriented programming is a very wide subject. Here, we just define the core concepts that are relevant to our analysis of object orientation in a distributed environment. Additionally, the implementation of object oriented concepts varies highly between programming languages. We attempt to present them in a language neutral manner, but still draw examples from certain archetypal languages. Our target application area of distributed systems is not restricted to a single language either, even within a single system. We will study how principles and implementations of inter-process communication relate to these generic object-oriented concepts in Chapter 3.

2.1 Abstraction

In procedural programming, algorithmic solutions to problems are constructed from *procedures*. These smaller program fragments perform sub-tasks in a "black box" fashion. Input data is passed to procedures as *parameters*. The parameters are often constructed from *results* which have itself been produced by procedures invoked earlier [BBG+63]. Recursive composition of ever higher level procedures leads to a complete solution to the original problem, as illustrated in Figure 2.1. Thus, procedures act as *functional abstractions* [LZ74]—building blocks that solve a particular task without their user having to know *how* exactly they do that.

The SIMULA 67 programming language introduced the concept of *classes*. Classes bundle together patterns of data and procedures operating on it (*actions*) [DMN68]. These

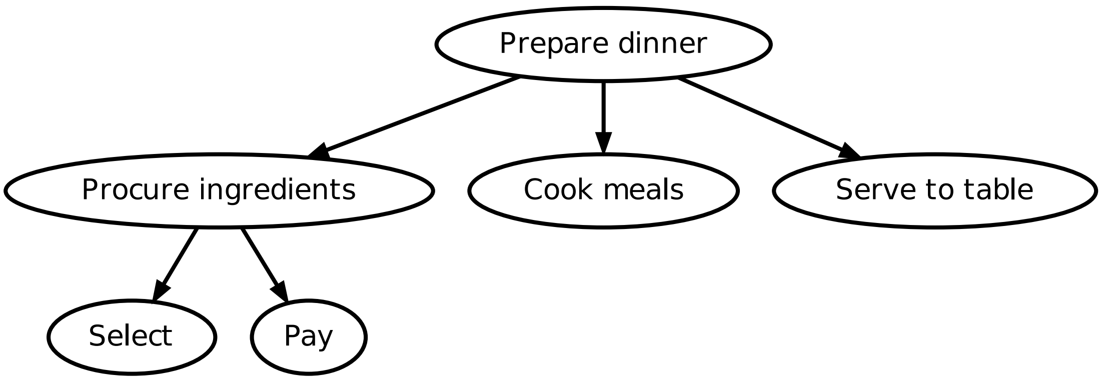


Figure 2.1: Recursive decomposition of a problem

collections of functionality can be used without manually transferring data between individual procedures. This makes them ideal as abstract reusable models of problem oriented concepts, such as customers, shapes or bank accounts, as opposed to a bunch of unrelated memory locations (variables) and executable code.

Classes are not live instances of the entity they are modeling by itself, but instead act as static templates, cookie-cutters for an arbitrary number of *objects* which fill this role at runtime [Coh84]. Objects of a given class all share the set of functionality defined in the class, but the internal data used by their member procedures is separate. This individual "memory" of objects is called their *state* [Weg87]. For example, there could be a `Person` class, which models real-world people. One object of this class could carry a birth date of `1948-05-12` as part of its state, while another one could simultaneously store the date `1963-05-04`. These objects are depicted in Figure 2.2 along with an object of another class.

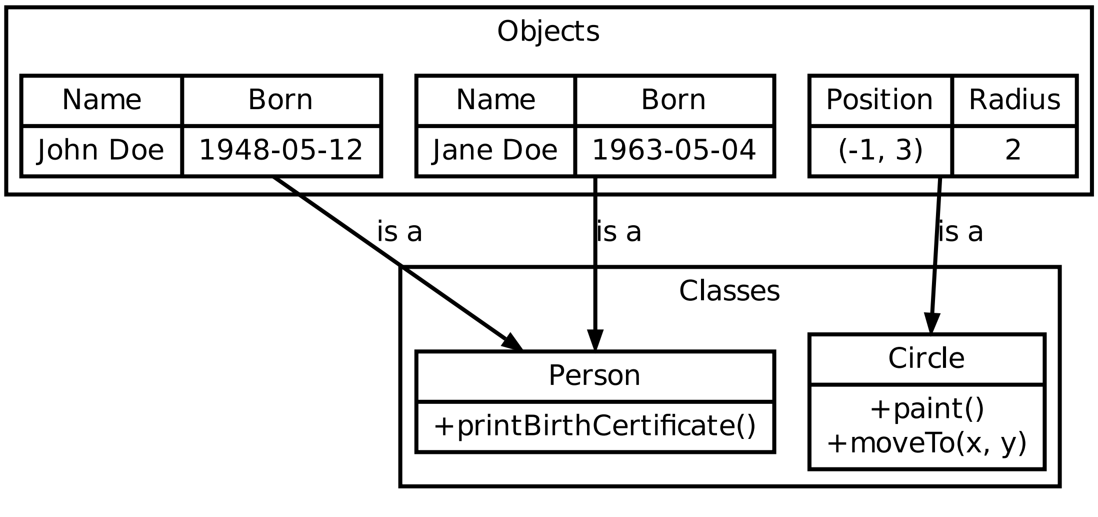


Figure 2.2: Objects and classes

Object state is represented by *member variables*. The data members of SIMULA objects can be directly accessed anywhere ([DMN67], Chapter 7). In fact, it's possible to usefully declare pure data classes that contain no other actions than access [DMN68]. Liskov and Zilles argued [LZ74] that the abstract semantics and available operations should define a data type, not the internal representation of its state. Accordingly, the principal difference between their concept of *operation clusters* and SIMULA classes is that cluster state variables are only accessible within operation implementations. The state is observed externally using inspection operations, which decouple the semantics of the data from its representation. Hence clusters provide a form of *data abstraction*, unlike SIMULA classes [Weg87]. The language used by Liskov and Zilles for their research was later named CLU, specifically after the cluster feature.

The use of the `class` keyword from SIMULA has been widely adopted in more recent object-oriented programming languages, such as C++. In many of these languages, outside access to object internals can be restricted [Sny86]. Thus, the classes of these languages have the data abstraction capabilities of CLU clusters, unlike those of SIMULA.

The key feature introduced in CLU that enabled full data abstraction was the *constructor* (called "create-code" in CLU). The constructor is a special class/cluster operation which initializes the internal representation of an object when one is created. It decouples the layout of the object from the parameters, if any, required by it to generate the values for the members. It is thus possible to change the internal representation of objects without breaking existing user code creating them, as long as the constructor can initialize the new layout from the old parameters. In the example in Figure 2.3, the way of storing a geometrical object's location could be changed from separate coordinate values to a `Point` class, while keeping the same external interface.

Simple operations which allow inspecting a particular aspect of an object's state are called *getters* in programming parlance (e.g. [Hug02]). Liskov and Zilles identified the need to perform calls to these small operations as a potential performance problem. Their solution was a compiler optimization, which replaces the calls with *inlined code*. Inlining means copying the operation implementation to the call site, such that function call overhead is avoided [LZ74]. However, as we will explain later, it is not very straightforward to realize this optimization in an inter-process context.

2.2 Inheritance

SIMULA classes could already be used as "prefixes" in each other, such that a class would *inherit* all data and actions from the classes declared as its prefixes [DMN68]. A class that inherits members from a prefix class is called a *subclass*. New members declared in

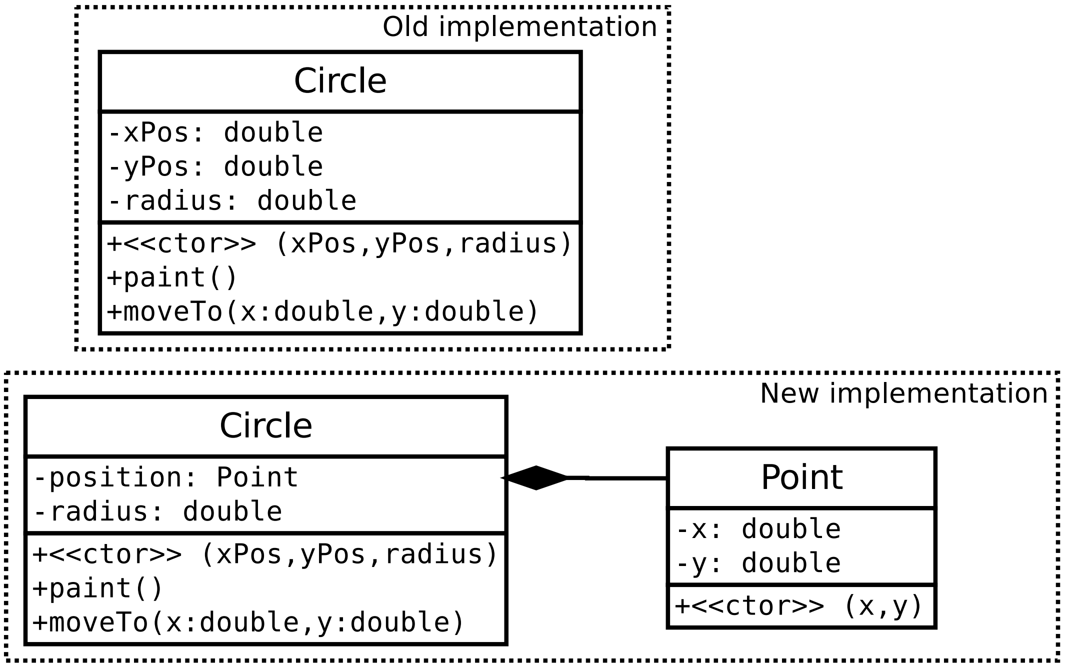


Figure 2.3: The constructor decouples the parameters used for creating an object from its internal layout

the subclass make it a conceptual specialization of the prefix class (also called *superclass* or "*parent*" class). For example, there could be a `Point` class which models a point in a two-dimensional plane. A `ColoredPoint` class could use it as a prefix, but add a color attribute. Thus, all `ColoredPoints` could be considered `Points`, but not the other way around. This example is illustrated in Figure 2.4.

Class inheritance can be applied recursively. The ”grandparent”, ”grand-grandparent” etc. classes which result are collectively called the *ancestors* of a class. Data and actions from all ancestors in this chain are prefixed to the final class.

The relationship between a subclass and its superclass can also be described by saying that a subclass is *derived* from its superclass. Multiple subclasses can be derived from one superclass. This kind of inheritance leads to a tree hierarchy of the classes, where each class node has its superclass as a parent and any subclasses derived from it as child nodes.

Declarations for SIMULA classes can also include a `virtual` part ([DMN67], Section 2.2.3). The operations declared therein will be defined by subclasses. This is an instance of the general concept of *subtype polymorphism*: the behavior of a virtual operation depends on the subclass defining it, and so of the actual subtype of the object it has been invoked on. This can also be considered a form of functional abstraction: The base class declares what an operation should do, and the derived classes define how to do it.

A fairly natural extension to the concept of inheritance is being able to derive a sub-

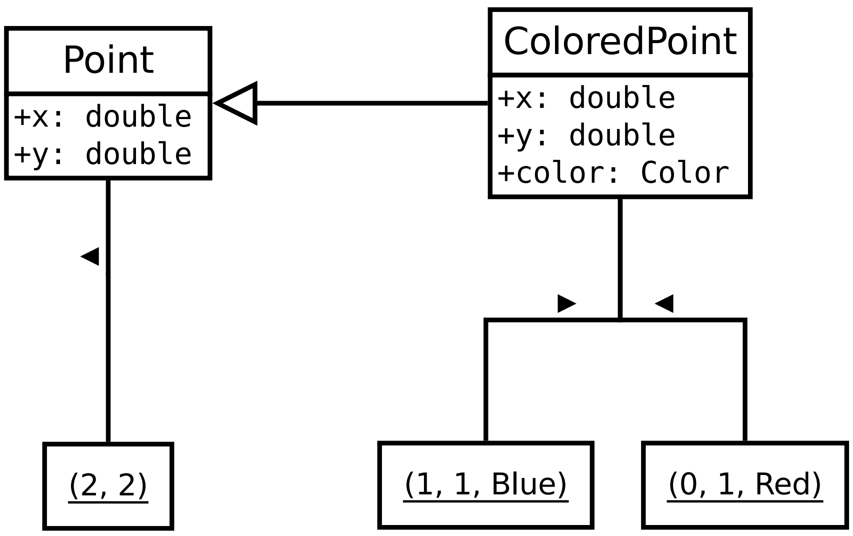


Figure 2.4: Specializing classes through inheritance

class from multiple superclasses simultaneously. This is called *multiple inheritance*. In contrast with the previously presented concept of single inheritance, the classes in multiple inheritance schemes form a directed acyclic graph topology, as shown in Figure 2.5. Multiple inheritance complicates subtype polymorphism somewhat: As it is possible for multiple classes at the same level of the inheritance hierarchy to define a common operation, there is ambiguity in which definition should be used [Sny86].

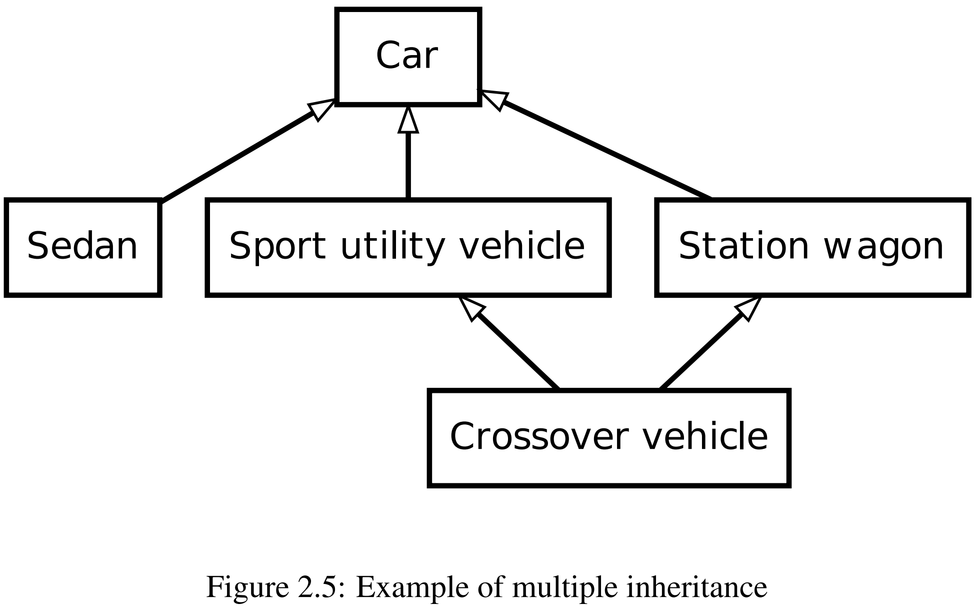


Figure 2.5: Example of multiple inheritance

2.2.1 Implications on typing

Liskov and Zilles suggested [LZ74] that languages should treat mismatches between actual and excepted types either as errors (*strong typing*) or through automated type conversions. This should happen for both built-in and user-defined (class) types. However, the cluster concept in their research language did *not* support any form of inheritance.

The introduction of inheritance complicates the notion of typing. What should happen when a derived type is used in a context which expects an object belonging to its parent class? Such a situation might occur for example when an object reference is passed as a parameter to a function.

In many languages, using a subclass instance where one of its parent class is expected is considered acceptable: In them, class inheritance directly establishes a *subtyping* relationship. Snyder associates [Sny86] these semantics with a thinking where the purpose of inheritance is recursive definition of types by specializing the parent class. From this perspective, the relationship between the parent and the derived class is a public commitment. Snyder further argues that it should be possible to use inheritance to facilitate *code sharing*, without making such a constraining decision. This is called *implementation inheritance*. Changing a class used as an implementation helper is guaranteed to not break existing callers only if implementation inheritance is not visible in the external interface of classes [Wol92].

2.3 Delegation

Objects are the runtime instances of classes. The runtime state of objects can change independently from other instances of the same class. However, the set of operations which defines an object's interface to its users is tied to what was declared in its own class and ancestor classes. In other words, the set of supported operations is *static* at runtime. Even more importantly, the specific virtual method definition to use for those operations is also chosen based on the inheritance relationships established when writing the program.

Delegation [Lie86] is the act of an object operation implementation to transfer the responsibility for its completion to some other object. The choice may be based on a runtime attribute, e.g. a reference to the object which is delegated to (the *delegatee*). This enables *dynamic* operation specialization, even for a single object instance, such as how instances of the `Car` class delegate the act of starting up to different `Engine`s in Figure 2.6.

Let us consider the parts of objects contributed to them by each ancestor class as separate objects in their own right. Then, we can interpret class inheritance schemes as defining a static delegation setup between these objects (e.g. Wolczko [Wol92]). When an operation is invoked on an object of a derived class type, it uses the derived class imple-

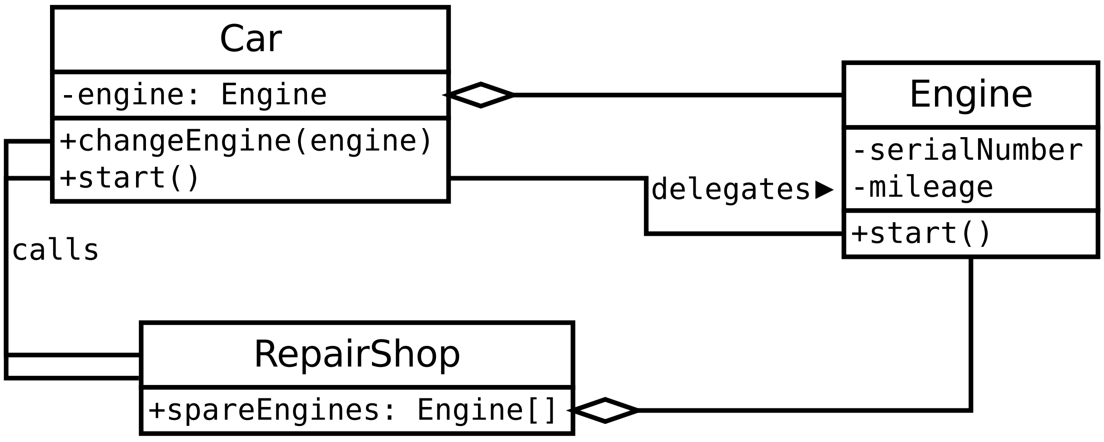


Figure 2.6: Delegating to an object based on a runtime reference

mentation, if there is one. Otherwise, it delegates to one of the part-objects corresponding to the ancestor classes. This is the case for the `moveTo()` operation in the example in Figure 2.7. This operation is the same for regular and colored points, so colored points use the implementation from their superclass. We can conclude that the concept of delegation is at least as general as that of class inheritance.

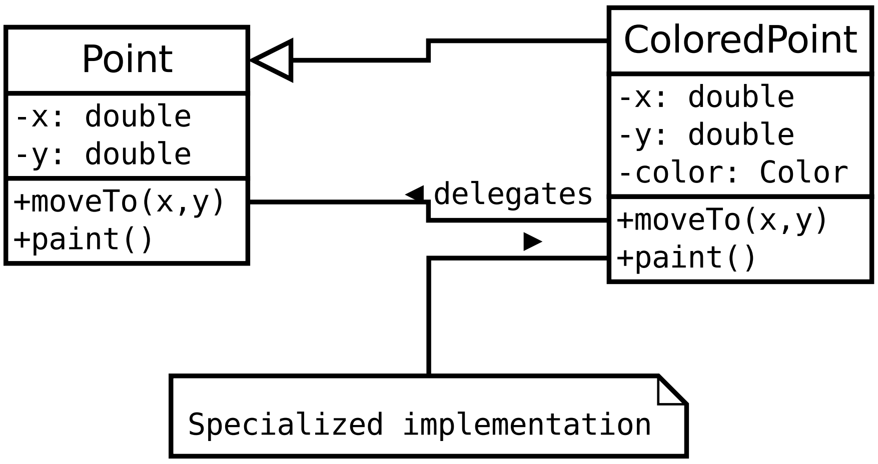


Figure 2.7: Delegating to an ancestor class

Central to the delegation pattern is an assumption that the delegated-to objects will often need to query the object delegating to them (the *client*) for additional details required for completing the task. Lieberman [Lie86] gives an example of a turtle delegating the responsibility for drawing itself to a pen. The pen must query the the turtle for its position to determine where to draw.

Wolczko [Wol92] demonstrates this side of delegation with an analogy of a manager giving a task to his or her subordinates. The subordinates consult the manager for details as required. It could be theoretically possible for the manager to specify all the details along with giving the task. However, this would imply specifying more details than needed for any given subordinate, as different employees will need different details to supplement their own knowledge. Only if the manager had perfect knowledge of the background of each employee, could he pass on just the relevant hints. But then the subordinates would not be treated uniformly any longer: there would be increased coupling between the manager and the subordinates.

Accordingly, it is a key feature for systems where general delegation is important to provide a mechanism for a delegatee to be able to query arbitrary additional information from their client [Weg87]. One such mechanism is *rebinding self*, which makes the delegatee dynamically able to refer to attributes of the current client object like they were its own. Wolczko notes [Wol92] that this is similar to how operations defined in parent classes can access attributes that might be redefined by the subclasses. However, because that is based on the static inheritance relationship, it is less flexible than dynamic rebinding to an arbitrary client.

With any mechanism for querying additional information from the client, it is necessary for the client to be able to respond to the query, although it is at the same time waiting for the delegatee to finish the operation. A deadlock would otherwise occur [Lie86]. This observation will be very important for us later in Section 3.5.2, where we define certain constraints for the messaging patterns used for communication between objects in different processes.

2.4 Explicit Interfaces

If class inheritance is used just for implementation code-sharing, superclasses should not be visible to the users of a subclass. This however implies that the public operations of the superclass must not be available either. Hence polymorphic behavior through a superclass reference is not attained.

The problem with inheriting both externally visible interface attributes and implementation from a superclass is that these two components are tied together. When deriving from a superclass to gain an interface compatible with its clients, one will also restrict themself to the overall implementation pattern of the superclass. On the other hand, when inheritance is used to reuse implementation, one will have to continue using that particular implementation helper class forever. Otherwise, compatibility will be lost with any users that depend on the parts of the external interface contributed by that class. Whatever the motivation for using inheritance is, unless the classes involved are very minimal, some flexibility has been sacrificed unnecessarily.

This flexibility can be regained by decoupling the implementation and external interface aspects of inheritance from each other. Recall that the external interface of a class is defined by the operations it supports. An *explicit interface* is a named collection of operation declarations without any associated implementations. By implementing the member operations of an explicit interface, a class can provide an interface for external access without placing any constraints on the internal implementation [CCHO89].

There are attractive properties in referring to objects by *interface references*—just having knowledge of exactly one interface they implement. One can call interface methods, and the object pointed to determines the class from which an implementation is used. Thus, calls through an interface reference are polymorphic. The actual class type of the object itself is not visible through an interface reference, so neither is what the class is derived from. Thus, the class can use whichever implementation helper(s) it pleases to implement the interface—either through class inheritance or delegation. Also, the object can implement an arbitrary number of additional interfaces without the user of a single *aspect* of it gaining knowledge of that fact. Hence, users can not accidentally grow dependencies to these other interfaces. This is illustrated in Figure 2.8.

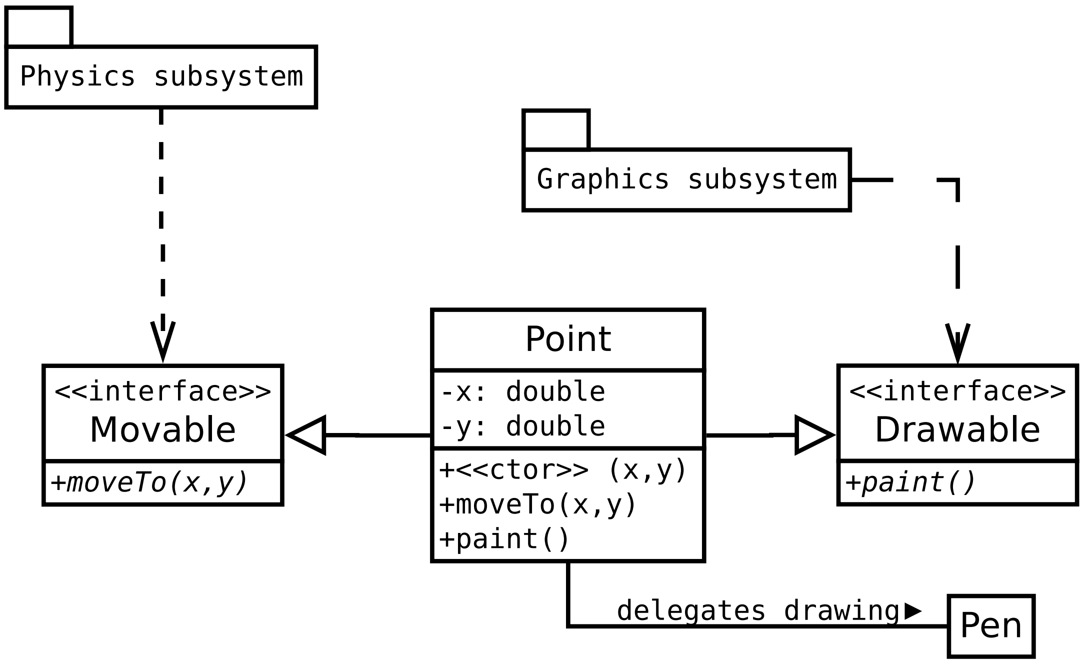


Figure 2.8: Accessing an object through an interface reference

Comparing objects for equality is a common operation. However, objects of different types can not be sensibly compared, even if they are derived from the same superclass. This is because specialized subclasses have additional information, which must be considered for a full comparison, but is not present in instances of “sibling” subclasses or the parent class [CHC90]. For example, the intuitive definition of equality for two objects of the `Point` class of Figure 2.4 from Section 2.2 is that their coordinate values match. However, is the blue `ColoredPoint` at $(1, 1)$ equal to a non-colored `Point` at the same location? This is an excellent example of how such a fundamental and widely applicable operation can cause conceptual issues if included, via inheritance, in the external interface of a subclass. One should not *pollute* interfaces that are intended to be generic with such ambiguous operations.

Interface pollution is actually an even more general issue. Objects are typically used by more than one kind of a client. The variety in clients naturally leads to differing requirements on the operations supported by the object. Let us consider one kind of client, which is intended to use a certain subset of the object’s public interface. If the interface visible to the client exposes aspects of the object interface outside that subset, the client might start accidentally depending on properties which are not really relevant to it. This danger constrains changes to these parts of the object, even if their actually intended users could cooperate in going through some needed modification. From this point of view, parts of the interface beyond the subset intended to be used by a given client are pollution. This leads to the *Interface segregation principle* [Mar96], which states that there should be a separate explicit interface for each kind of a client. Turning the relationship around, there should be an interface corresponding to each *role* the class is supposed to “act in” for its clients.

In the manager and subordinate example of the previous section, the manager had a role of answering queries from subordinates who need additional information. He will likely also have a separate role in which he has to report the progress of tasks to his own superiors. These could be formulated as a task detail query and as a task status reporting interface, respectively. This increases flexibility: If the manager started using some kind of automated reporting software, he could pass on a reference to the reporting interface implemented in this software system to his superiors instead of having to respond to report queries directly himself. Thanks to the explicit reporting interface, he can do this safely. There are no worries that the superiors might have accidentally carried away to snooping around task details. This info would not be available through the software their reference (say, an email address) now points to, but only to subordinates. This isolation scheme is shown in Figure 2.9.

The reporting software used by the manager is a good example of an additional construct closely related to interfaces. Whereas interfaces are collections of operation dec-

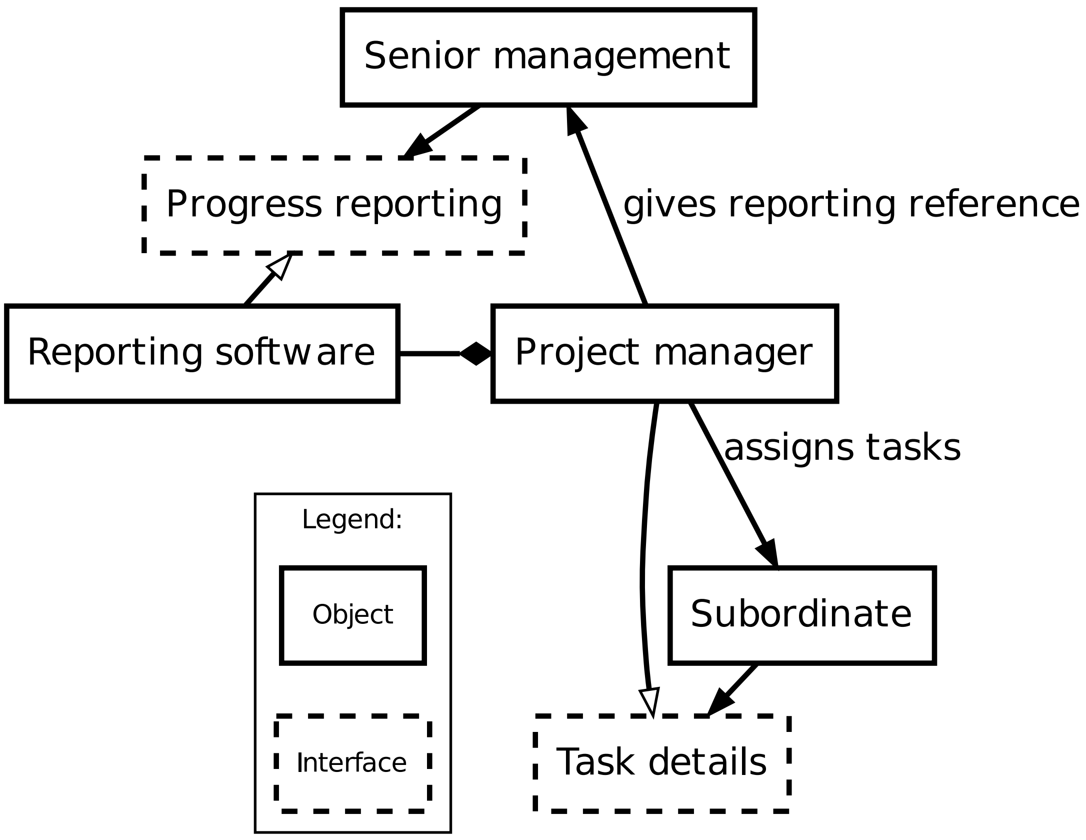


Figure 2.9: The possibility of implementing an interface transparently using a helper object

larations, *mixin classes* [BC90] are bunches of *definitions* for functionality. As the name implies, their functionality is intended to be "mixed in" via multiple inheritance or delegation to produce composite objects. Mixing in to an arbitrary object is only possible if the definition of a mixin class does not depend on knowledge of the other classes mixed in with it. This requirement is very similar to the flexibility rationale behind the interface concept. If there is a sufficiently close mapping between external explicit interfaces a class is required to implement and the mixin classes available to it, the class can use the mixins as swappable implementation helpers for the interfaces (e.g. [Ber00]).

2.5 Design by Contract

The practice of *defensive programming* advocates guarding all operation implementations against misuse and promises broken by the operations they themselves invoke. This is to be done by explicitly checking that all assumptions made by the code hold true. This leads to callers checking that everything is in an allowed state for calling a method, and the called methods itself checking that the caller did this preparatory work. Meyer argues [Mey92] that these checks are mutually redundant, and hence add unneeded complexity. This makes the checks counterproductive—their purpose is to increase reliability, but according to him, that has an inverse correlation with complexity.

To remove the intrinsic redundancy from defensive programming, while retaining its benefits on special case survivability, Meyer proposes a division of the checking effort. Methods must document all of the conditions that must hold true to make them usable (*preconditions*), and the additional properties they make true when successfully called in a suitable state (*postconditions*). Before calling an operation, a caller ensures that all of the preconditions hold true. Then, the called operation can safely assume that the preconditions are fulfilled, without explicitly checking for them. Similarly, the caller does not need to check that the operation did what it was supposed to, because this is guaranteed by its postconditions. Meyer relates this mutually beneficial agreement with real-world contracts between clients and suppliers. As a consequence, this design ideology is called *design by contract*.

In Section 2.1, we introduced the constructor, a special class operation responsible for initializing the member variables of a created object. Thus, a postcondition of the constructor is that the internal state of the object is consistent. The consistency relations established by the constructor are collectively called the *class invariant* [Web01].

As a practical example, a container object must have at least as much space reserved as there are elements. A constructor that pre-fills the container with a certain number of elements (e.g. from another container) must therefore take care that this condition is fulfilled after its execution finishes. For operations adding elements to the container, there are two design alternatives. Either we have a separate resizing operation, which must be called by the user explicitly to guarantee that there is sufficient space for adding elements, or the insert operations do this automatically and have no such requirement. In the first alternative, the insert operations have a precondition that sufficient space has been reserved beforehand. In the second option, automatic expansion to a large enough capacity is guaranteed by the class invariant, which all of the operations must preserve. These two approaches are compared in Figure 2.10.

Guaranteeing that the preconditions are fulfilled is mainly of use to an operation's concrete implementations, which can omit explicit special case checks. Similarly, postconditions impose specific requirements on what the implementations must achieve. However, the contract they together form is part of the operation's abstract *declaration*, not specific to a given implementation [Mey92]. Any implementation of an abstract operation must fulfill the terms of the contract for the operation. However, an implementation can of course do even better—that is, require their caller to ensure less preconditions, or make

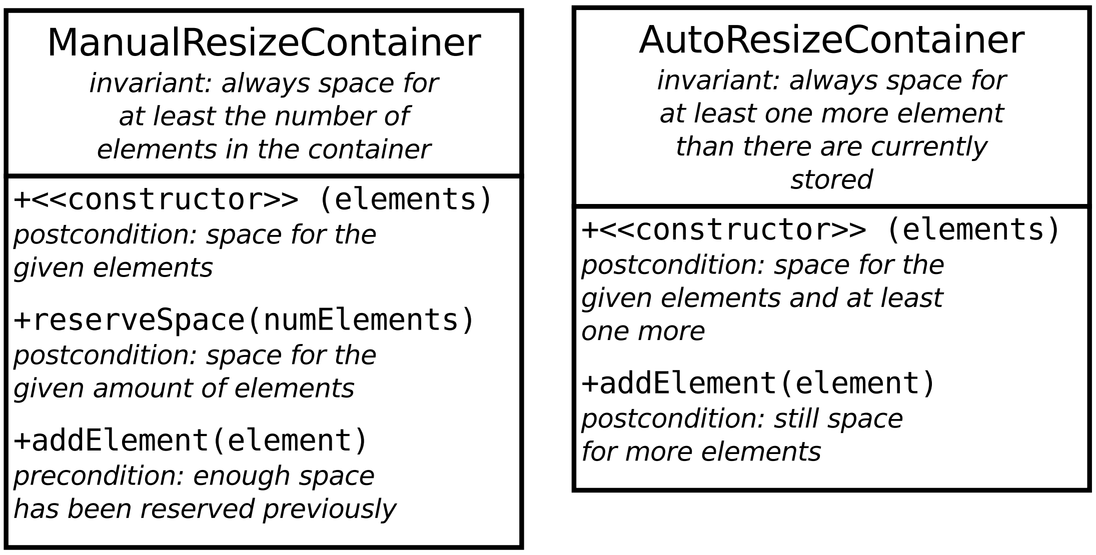

ManualResizeContainer
invariant: always space for at least the number of elements in the container
+<<constructor>> (elements)
postcondition: space for the given elements
+reserveSpace(numElements)
postcondition: space for the given amount of elements
+addElement(element)
precondition: enough space has been reserved previously
AutoResizeContainer
invariant: always space for at least one more element than there are currently stored
+<<constructor>> (elements)
postcondition: space for the given elements and at least one more
+addElement(element)
postcondition: still space for more elements

Figure 2.10: The class invariant and method contracts

more postconditions hold than required. Meyer makes this observation in the context of overriding superclass method implementations. However, it is equally applicable to explicit interfaces, as operations are declared in them with the same level of detail as in concrete superclasses. Thus, the contracts established by an explicit interface must be followed by all objects implementing it. Otherwise, users of the interface would need to separately consider the requirements (preconditions) imposed by each possible implementation. If this was required, the flexibility offered by the decoupling the external interface from its concrete implementation would be lost.

2.6 Exceptions

It is not always possible for a method implementation to fulfill its contract, such that even though its preconditions have been satisfied, the postcondition can not be achieved. This is because the preconditions can never be truly exhaustive. For example, a finite global resource, such as working memory, might be depleted between invoking an operation and the operation trying to allocate some of the resource for use. Also, regarding this specific example, very few systems provide any means for a caller to check for sufficient available working memory in the first place. When a procedure encounters a *force majeure* event like this, it must somehow tell its client that it failed.

There are a few possibilities for reporting errors to the user of an operation. A simple approach is to extend the result domain of the operation with additional values that indicate failure cases. As an example, consider a function that returns the square of a real number ($x \mapsto x^2, x \in \mathbb{R}$). The result of this operation is non-negative for all $x \in \mathbb{R}$. If a real number type which can represent both positive and negative values is used for the return value of this function, negative values can be returned to indicate errors in the calculation, such as arithmetic overflow. This approach is however problematic, first, because for some operations the entire range of values representable by the return type are needed for actual results. And second, because in this approach essentially the same event (e.g. memory exhaustion) has to be represented in different ways by operations with dissimilar result domains. Additionally, there is a danger of the client interpreting the error code as a normal result, or ignoring the result completely.

A better way to signal errors, provided the programming environment supports it, is to raise an *exception* [Goo75]. An exception is a uniform way to indicate that a particular erroneous circumstance has been encountered, no matter what the result domain of the operation is. Furthermore, the programming environment can enforce the calling function to take some action when an exception is raised, or failing that, perform a default action such as program termination. The rationale for *handling* the exceptions in the callers of the operation which encountered the error is that it they have the best knowledge of the big picture. This knowledge can be used to gracefully recover from the error, or consciously ignore it.

Goodenough's example [Goo75] of an error which should be handled differently depending on the high-level context concerns an input stream, perhaps one that reads data from a file. For a byte-by-byte read operation on the stream, encountering the end-of-file position is a critical error. However, for a higher level procedure, the purpose of which is to read a file in its entirety to memory, this event would simply be an indication that its task is finished. This is different for an operation that reads multi-byte records using the same byte read operation. For it, only managing to read half of the bytes that are needed for a complete record before the error occurs would be a failure.

So, external circumstances might prevent an operation from finishing successfully even if its formal preconditions were fulfilled. But what if they were not fulfilled in the first place? Meyer notes [Mey92] that in a single-threaded, local process context, this is always a programming error, because the client has failed to keep its side of the contract. However, he also observes that this is different when multiple threads access an object concurrently. Then, a caller might have properly ensured that a precondition holds true, but a parallel thread may cause it to turn false again by the time the invoked operation depends on it.

Another kind of class invariant violation might occur because of the fact that operations are usually sequences of multiple subtasks. Class invariants may be broken by the first steps, as long as they are re-established later, before the method finishes executing. Thus, a parallel or re-entrant invocation of one of the object's operations between these steps might find that the class invariant is not satisfied [Web01]. An example of this is depicted in Figure 2.11. We will compare a few patterns of inter-process method invocation considering these aspects in Section 3.5.2. For now, it suffices to observe that in the presence of multiple concurrent callers, the fulfillment of preconditions and class invariants is not a given. Thus, gracefully handleable run-time exceptions are needed to communicate when they have been violated.

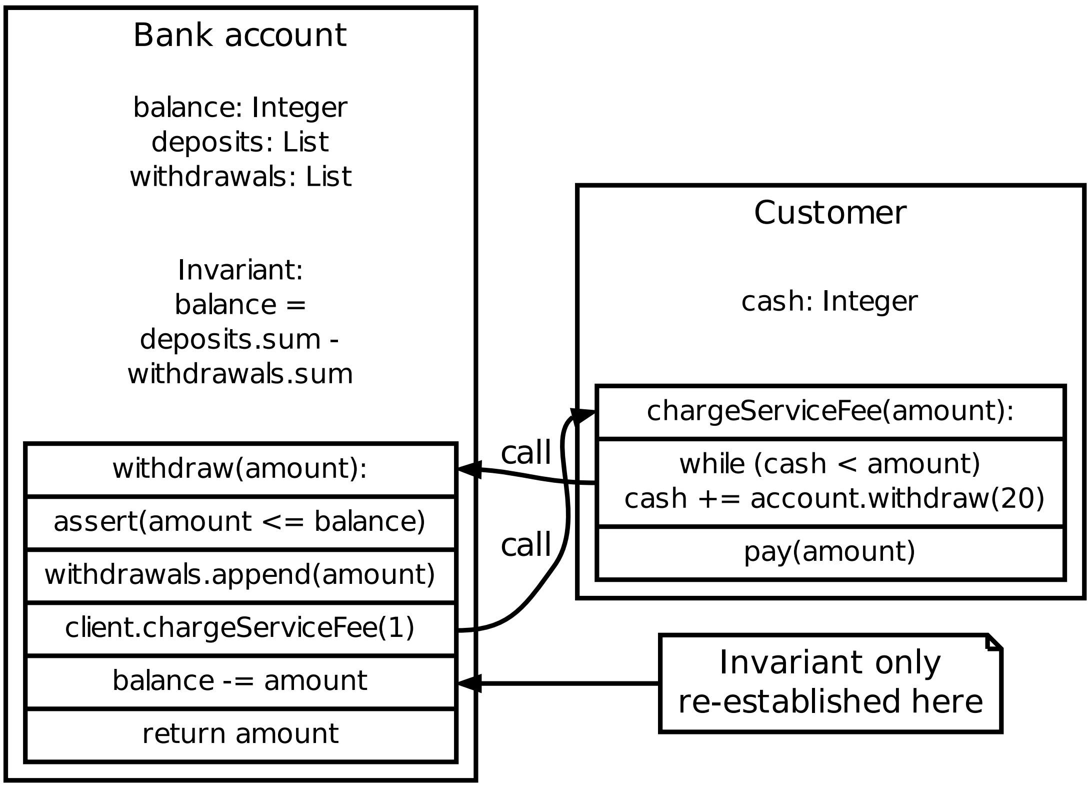


Figure 2.11: The invariant of the account class is broken when the withdrawal is recorded, which is done before the call to charge a service fee. The invariant is only re-established after the call, by adjusting the balance. Thus, the invariant is not satisfied during a possible re-entrant method invocation.

2.7 Events

The previous sections have been concerned with structures combining operations with data, building hierarchies of such structures, and various ways to hide the implementation of their operations from the callers. However, even with these mechanisms, the caller still always *explicitly invokes* a particular operation, with specific arguments, on a particular instance it has a reference to. Thus, the caller is strongly coupled to the particular interface it uses.

An alternative mechanism for object interaction is *implicit invocation*. In this paradigm, objects *announce events*, which cause handler methods to be invoked on other objects. The announcing object does not need to know which methods on which objects are invoked, or if any are invoked at all. This new kind of *decoupling* enables hooking up new components to be invoked by old ones, which do not need to be modified [GN91]. Furthermore, making different kinds of event-handler connections between the same components allows altering the system's behavior without any changes inside the components. This makes it possible to reuse the components in varying contexts with reduced effort [XZ02].

The flexibility benefits require that an event announcer does not need to know about the *listeners* for its events. Thus, there must be some kind of separate registry of connections between announcers and listeners. As illustrated in Figure 2.12, this subsystem invokes the correct handlers at runtime for each announced event [XZ02]. There are multiple ways to implement this connecting service even in the context of a single process. These range from restricted mechanisms integral to the language itself like the Smalltalk-80 model-view-controller facility, through language extensions provided by a preprocessor, to library facilities implemented using the target programming language itself [NGGS93]. In Section 3.3, we will study a similar concept for distributed systems. Then, in Section 3.4.3, we will analyze the bus daemon of the D-Bus message bus system as a practical implementation of the connecting service for a multiple process context.

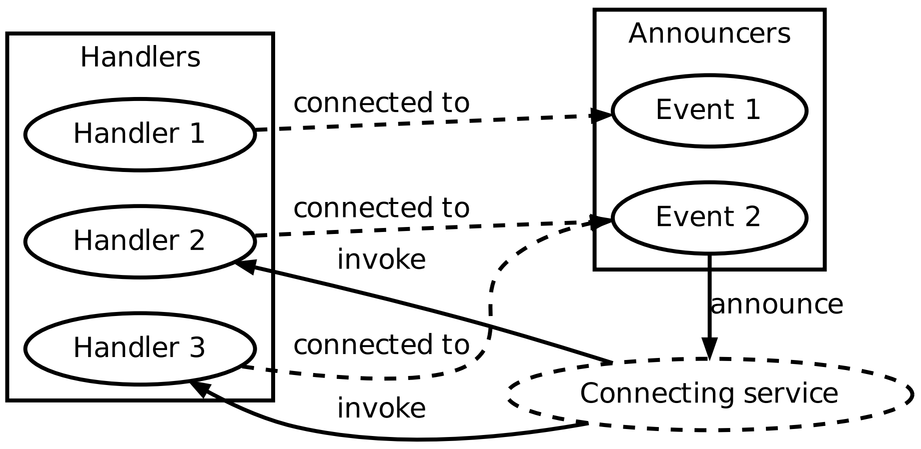


Figure 2.12: Connections between events and handlers are registered with a mechanism that is independent of the event announcers

The capabilities of implicit invocation systems vary due to constraints set by the programming environments and the applicability to the general style of programming in them [GS93]. This variability includes dimensions like where and how events are defined, and whether they can be defined dynamically or just at compile time. The same goes for connections between events and handlers, and how closely any statically defined parameter lists of the events need to match those of the handlers connected to them. Although flexible dynamic connections decrease the predictability of the system [NGGS93], we view that this capability is essential due to the increase in reusability it offers. In fact, it would be completely impossible to implement our extended proxy construct without dynamic event handler registration, as we will see in Chapter 5.

The existence of the connecting service enables an event announcer to avoid explicitly invoking the handlers. However, parties that register connections between these must still explicitly specify the events and the handlers. A handler is a method of some object, which in this use fulfills the role of a listener for an event or a set of events. Thus, Martin's Interface segregation principle (ISP) (Section 2.4) applies, and for maximum reusability, the listener should be referenced through an interface specific to this particular listener role.

Additionally, just like handler methods are a part of the listener objects' external interface, so are events a part of the publicly visible interface of announcers. In fact, exported events and methods can be considered equal contributions to an interface [XZ02]. We argue that the ISP should apply to the interfaces where events are declared just as it applies to those with methods. It should however be noted that events seldom constitute a fully independent role. For example, an event might be announced when a buffer object has space for inserting more elements. It would be a sensible middle ground to have both this event and the insert operations in a role-based interface used by clients that insert items. This arrangement is illustrated in Figure 2.13, complete with another interface to be used by clients that extract items.

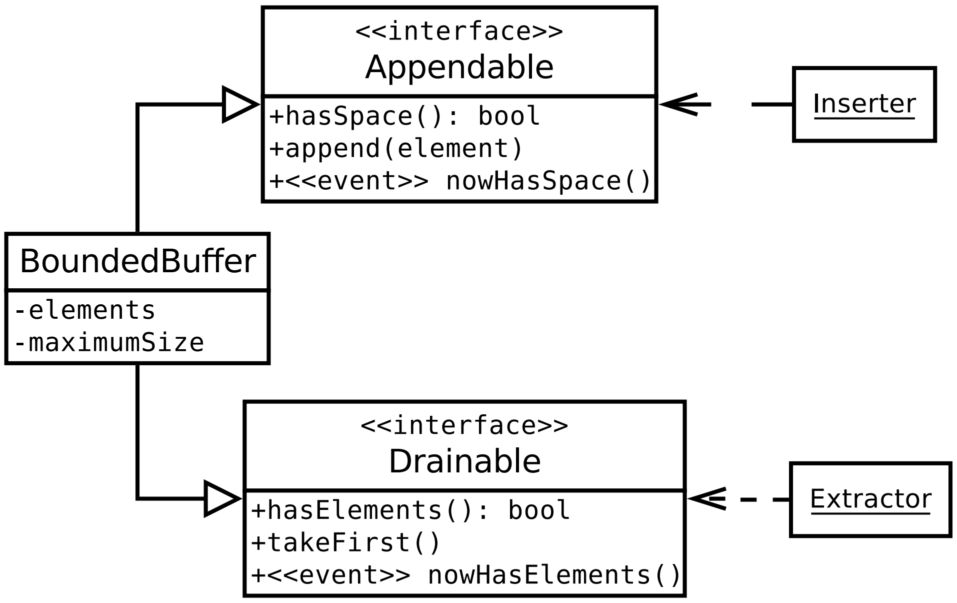


Figure 2.13: Example of role-based distribution of events across interfaces

3 INTER-PROCESS COMMUNICATION

Functional abstractions and encapsulated objects enable applications to delegate solving certain subtasks to self-contained procedures and modules. These building blocks can be put to shared libraries to increase their reusability. However, they still need to be loaded to the application's virtual memory space and executed locally. In distributed systems, it is also possible to invoke functionality from other *processes* running on the same computer (*node*) and possibly even on other nodes [DSC92]. Here, we use the term *process* to refer to an operating system construct that comprises a single protected address space and a set of associated resources. One or many *threads of control* can run in a single process, each with their own instruction pointer.

For it to be possible for an application to invoke functionality and query data from other processes, there must be a method of communication that is able to cross the protection boundaries between processes in a safe and controlled way. This is the essence of *inter-process communication* (IPC). Figure 3.1 illustrates the concepts of nodes, processes and threads, and forms of IPC between them.

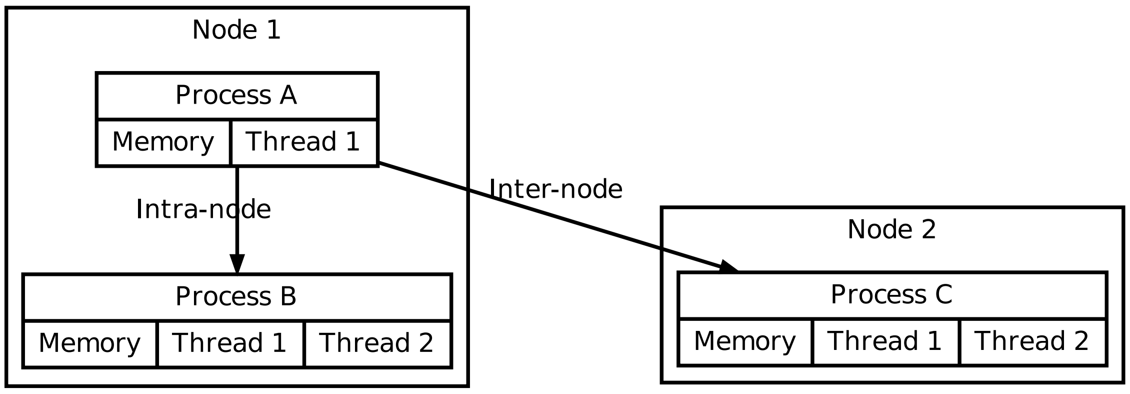


Figure 3.1: Nodes, processes and threads in a distributed system

One mechanism for inter-process delegation is to put the code and data to be used from multiple processes in a special *shared memory* area [Hor90]. This scheme is depicted in Figure 3.2. Using shared memory is especially workable within a single node.

In the shared memory approach, code is invoked by a direct call, like a normal procedure implementation that resides in private memory. Thus, restrictions are imposed on the implementation of the caller and callee. In particular, their code and data representations must be compatible. Hence, it is often not possible to call code that has been written in a different programming language, except for a small set of languages which can be used through a *foreign function interface*. Such interfaces can however be rather restrictive. And most certainly, it is not possible to share code compiled for different processor architectures or operating systems in this fashion—that is, code that targets heterogeneous

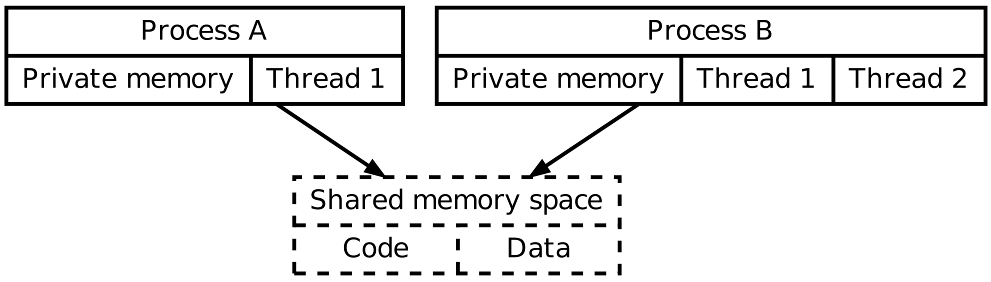


Figure 3.2: Code and data in a shared memory area

nodes. For general inter-node access, the call arguments and results must be *mapped* between the particular representations required by the two systems [BCL+87], or a single *system independent* representation must be used. The use of such a representation can also enable communicating with local processes that are implemented with different programming languages and runtime facilities, as long as they provide a compatible remote call interface.

Benefits of IPC within a single node

In the introductory chapter, the Telepathy framework for online real-time communication [dTdCDG+12] was used as an example of the benefits that IPC offers for desktop applications, even in the context of a single node. Here are a few further reasons for Telepathy's modular architecture [Har08]:

Robustness Components of the system can crash and/or be restarted individually.

License independence Licensing differences may restrict linking libraries to a single process. The multi-process approach allows some processes to be open-source and some closed as needed, but still function together.

Increased code reuse The language-independent distributed architecture forces defining abstract interfaces for inter-process access. As explained in Section 2.4, such interfaces enable transparently reusing a frontend (client) although the backend (service) is changed (e.g. to one implementing a different IM protocol). The same backends have also been successfully shared by separate user interface (frontend) implementations for different desktop environments.

Security A monolithic application requires privileges to directly access all of the resources that its functionality depends on. Each domain-oriented module of a distributed system only needs a smaller credential subset. An example of how this would apply to a system that handles various kinds of textual communication can be found in Figure 3.3. Furthermore, the interaction between the modules can be controlled and monitored by the messaging system that connects them together.

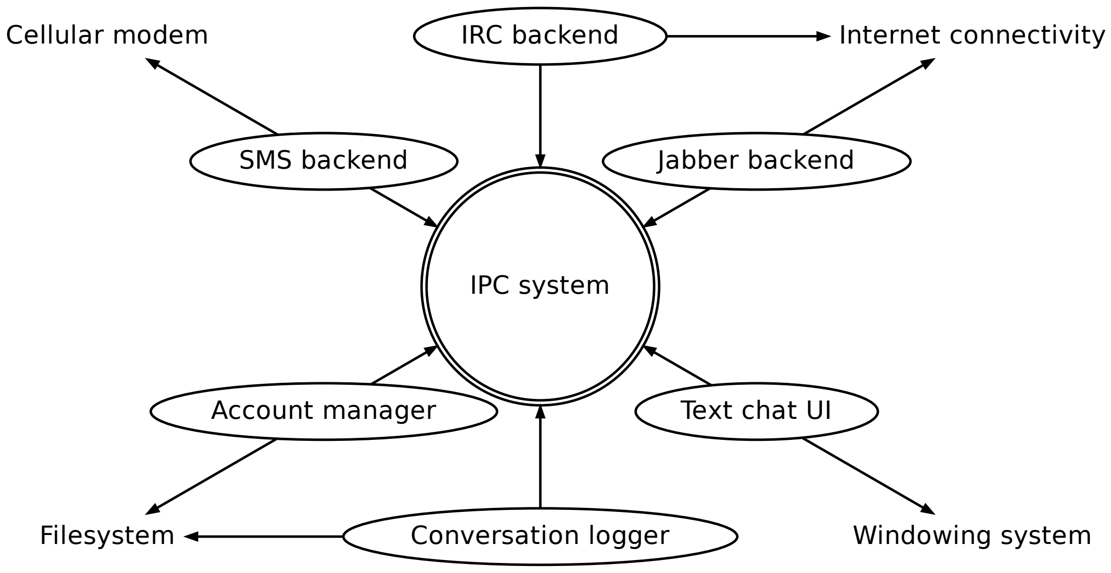


Figure 3.3: Distribution of privilege requirements over a modular application

Wittenburg identifies [Wit05] similar reasons for the components of a desktop communicating together. These reasons include sharing globally unique services between applications (like IM network connections in Telepathy's case), announcing events such as application start-up progress, and transparency on whether an invoked service is local or remote to the node. The observation that desktop services are *scriptable*, which means that they can be machine-driven, is shared with [Pet07]. In our experience, this capability has also been utilized in real-world Telepathy implementations.

In summary, there are many functional benefits to be gained from making desktop applications distributed, even within a single node. Real-world studies [BALL90] put the portion of inter-node access at just a few percent of all IPC. This might be because inter-node access is always much slower than calling a process in the same node. Hence, real-world software architectures tend to avoid overdependence on inter-node calls, wherever performance and/or interactivity is essential. Accordingly, for most of our research, we will focus on the inter-process communication technologies and patterns that are most suitable to the single-node context.

Concentrating on node-local inter-process communication allows us to side-step a few challenges associated with more general IPC, e.g. across the Internet. In particular, we do not consider the complex global *resource naming* and *location* services that are used for announcing exported services [RSW98]. Furthermore, access control is simple to implement in the local context with mechanisms such as UNIX credential passing ancillary messages [Lin10]. Network address translation (NAT) schemes and firewalls also significantly hinder inter-node access and peer-to-peer communication through the Internet in general.

The D-Bus messaging system is a main focus for the rest of this report. As shown later, it is especially suitable for IPC within a single node. There is less attention in its design for inter-node usage. However, the Telepathy Tubes [HOD+11] mechanism extends the reach of D-Bus over the Internet. This is accomplished by piggybacking on firewall/NAT traversal and service discovery mechanisms which already exist in instant messaging services for purposes such as file transfer, media streaming (Voice/video over IP) and presence announcement. Thus, the usability domain of D-Bus is extended, and so is the application area of the research presented here.

The first few of the next sections present general concepts of distributed systems, through literary review. This establishes context for the study of the D-Bus messaging system that follows. Finally, we will set out some guidelines for how messaging protocols should be designed when using D-Bus and similar systems.

3.1 Remote Procedure Calls

A low-level notion of interprocess communication can be based directly on transport layer network services, like byte stream or datagram sockets, or pipes [Wit05]. These transfer untyped data as essentially just sequences of bytes. When a client and a service are connected together using such means, they must obviously agree on the semantic meaning of the bytes. This agreed meaning constitutes the *network protocol* that is used in the communication. However, designing a byte-level protocol separately for each client–service application pair is cumbersome, and also steals implementation attention away from the actual unique functionality of the system.

In Section 2.1, we introduced the concept of procedural programming. In that paradigm, functional abstractions provided by *procedures* are used as central building blocks of algorithms. *Remote procedure calls* (RPC) invoke procedure implementations from other processes [BN84]. From the point of view of one writing an algorithm, classic RPC appears almost exactly like calls to procedures in the local process. In either case the implementation of the procedure is not important, only the task it accomplishes. Thus, the implementation can as well run in some other process if that is needed e.g. to gain access to a certain resource owned by it. The motivation behind this IPC paradigm is that programmers are used to procedural programming. Hence, emulating it in the distributed context requires minimal reorientation and relearning for creating distributed systems.

A key feature of RPC facilities is that they are not application-specific; the fundamental mechanism offered by them—a way to transparently transfer control to a remote process—can be used in building any kind of software where procedural programming is useful. By using RPC, low-level network protocol details do not need to be considered in client and server design, but merely the set of procedures the server exports for clients to call. Thus, RPC can be considered a reusable application level protocol for building distributed systems that follow the procedural programming paradigm.

For a remote procedure to be callable like a local one, there must be an ordinary local function with the same type signature as the remote one. Instead of performing the computation or other action by itself, this function causes the call to be *forwarded* to the remote process using lower-level RPC facilities. These forwarding procedures form the the *client stub*. The client stub type-checks and encodes the call arguments in a form that is suitable for transfer to the service. A corresponding *service stub* component in the remote process decodes the arguments. The stubs can do this because they have knowledge of the exact signature of the procedures. The stub code is often automatically generated from a description of the signatures of the procedures to export. The data transfer itself can be handled by a runtime library service that is common to all kinds of services and hence to sets of procedures. This layered architecture is illustrated in Figure 3.4.

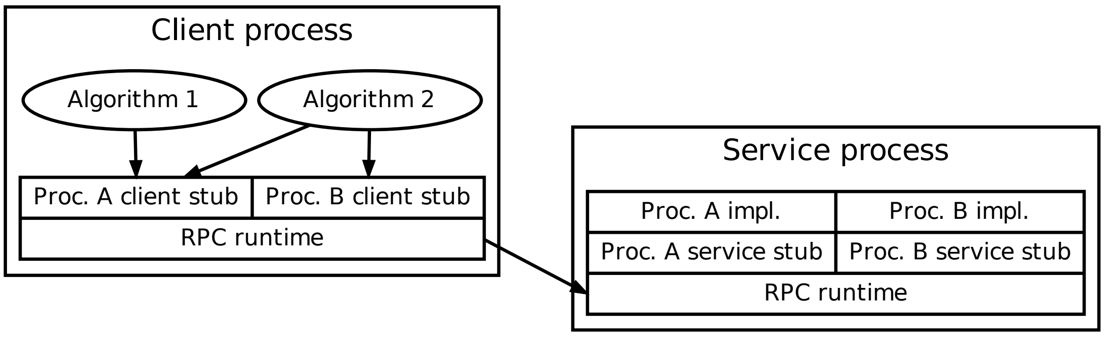


Figure 3.4: Client and service stubs in an RPC system

The RPC runtime can transfer the encoded call arguments to the remote process in a number of ways. The archetypal example is sending a *call packet* over a transport layer connection, as in the original implementation of Birrell and Nelson [BN84]. As mentioned in the introduction to this chapter, most inter-process calls do not actually need to be transferred over the network between computers, but access processes running on the same node. Intra-node calls can be optimized, e.g. by sharing the stack memory containing the call arguments between the processes. In this approach, the arguments do not need to be sent over a communications link, but they can simply be read from the shared memory by the service. This approach, illustrated in Figure 3.5, can yield a multi-fold speedup [BALL90]. However, it also hinders cross-language and especially cross-architecture communication, because the service must be able to interpret the data in the exact form the caller code passed it to the client stub.

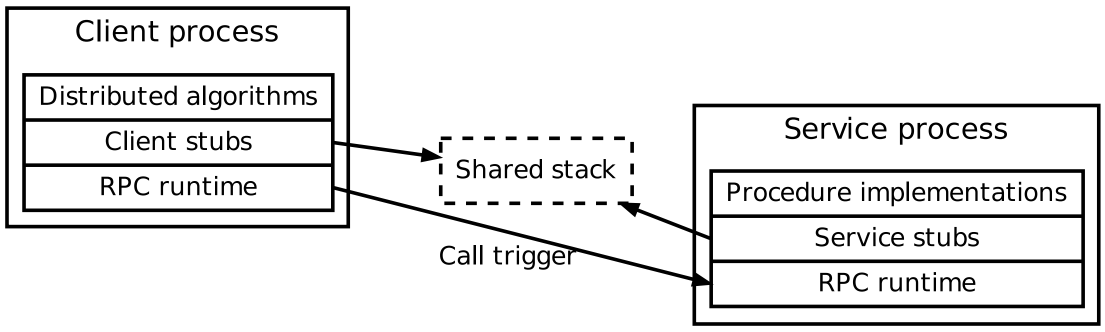


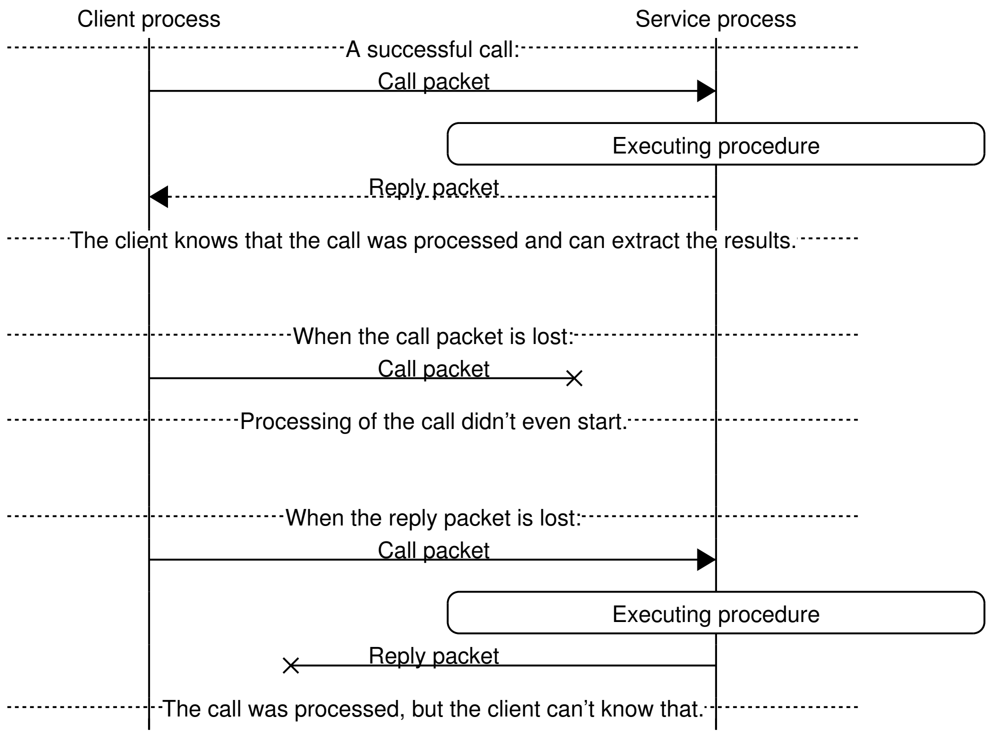
Figure 3.5: Sharing the argument stack between processes

Transparent mapping services between data representations and call semantics have been proposed to correct the cross-system communication issues in naive RPC mechanisms. The aim of this is to enable integrating existing heterogeneous systems together with minimal modifications. [BCL+87] The mapping adds overhead, and already implicitly constrains the type of information that can be passed to that representable in both the client and server systems. Based on these facts, we argue that it is better to just define RPC messaging semantics and a data syntax in a system independent fashion. As we will see later, this is also the approach taken by the D-Bus IPC system. The reimplementation effort is not a major concern, as in our opinion, IPC still has not been exploited nearly as much as possible in the area of desktop applications.

3.1.1 Asynchrony and return values

So far we have discussed just one side of RPC, namely that of *initiating* calls. The return values of procedures and the time at which their execution finishes are also significant in procedural programming. However, RPC mechanisms differ in their method return semantics. In classic RPC, the caller process is *blocked* until the service has informed it of the outcome of the call via the delivery of a *reply packet*. The reply packet is translated to the local stub procedure's return value, or an exception if the call failed. Only after this, client execution continues. Thus, the execution of the client and the service is effectively *synchronized*, so that only one of them is executing at a time.

In addition to errors encountered while executing a procedure, an exception can also result in RPC if the communication link between the client and the service process is severed [BN84]. This can happen not only before the service starts processing the call, but also during the execution of the operation. It is also possible that even though the procedure itself has been executed successfully, it will be impossible to tell that to the caller. Thus, the state of the service is ambiguous to the client after a communication error. A few related scenarios are illustrated in Figure 3.6. We will revisit this subject in Chapter 5 in the form of *invalidation*, when presenting our extended proxy construct.



Client process
Service process
A successful call:
Call packet
Executing procedure
Reply packet
The client knows that the call was processed and can extract the results.
When the call packet is lost:
Call packet
Processing of the call didn't even start.
When the reply packet is lost:
Call packet
Executing procedure
Reply packet
The call was processed, but the client can't know that.

Figure 3.6: Ambiguity of service state after a communication failure

The synchronous semantics that are inherent to traditional remote procedure calls do not allow client execution to continue in parallel with the service. Thus, even for raw computations which do not have to wait for any other resource to be available besides central processing unit (CPU) execution time, parallelism which could net a speedup in a multiprocessing system has been lost. Similarly, to execute some operations, external resources like network services, peripheral hardware devices or even the human user may need to be queried. Communicating with these resources can impose delays on the completion of an operation, even if there is idle CPU time in the system. If a process blocks to invoke such an operation, the use of CPU time may become inefficient even on a uniprocessor system. This is because the calling process might be able to utilize the idle CPU time for tasks such as animating its user interface [Pen08].

To regain parallel execution capability with synchronous RPC, multiple threads can be used in the caller process, as portrayed in Figure 3.7. However, the use of threads is programmatically unwieldy [ATK92]. The threaded approach is also not scalable to a large amount of outstanding calls, as threads with separately allocated execution stacks need to be spawned for the sole purpose of sitting waiting for a call to finish [WFN90].

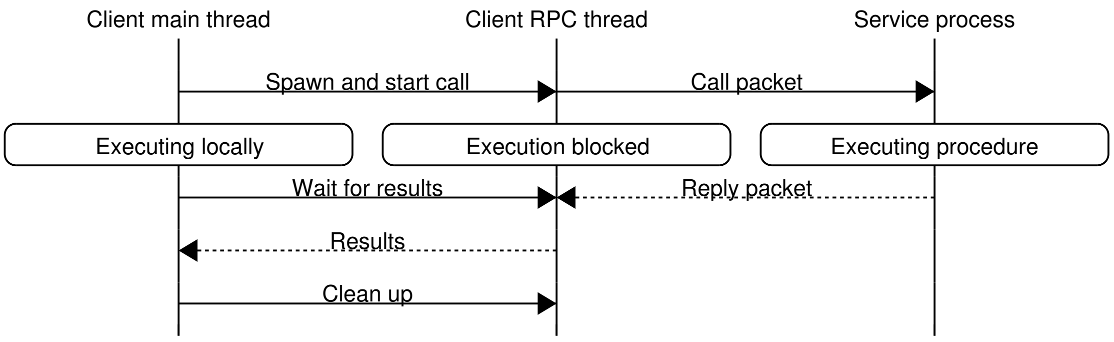


Figure 3.7: Using threads to continue local execution during RPC

An alternative to running synchronous RPC in threads is to make the remote calls itself *asynchronous*. This means that the calling thread is not blocked for the execution of the remote procedure, so it can continue to do local processing. In addition to the performance aspects described above, some correctness arguments on why inter-process calls must handled asynchronously will be presented in Section 3.5.2.

Some asynchronous RPC systems try to batch similar calls together to increase efficiency. This can go as far as to result in *call reordering* [ATK92]. Reordering means that a call that is made first by a client is actually executed later than a succeeding one. We will further elaborate on the problems associated with message reordering in Section 3.5.1.

Many asynchronous RPC systems do not actually have support for methods returning values to the caller. Additionally, the client is not notified of even successful execution of a procedure in some of them ("fire and forget" or *may-be* semantics) [ATK92]. This is different from local calls and synchronous RPC, in which there is universal support for determining the outcome of procedure invocations. Despite our asynchrony requirement, we stress that this bidirectional data exchange capability is essential for all IPC systems.

The incompatibility between asynchronous remote calls and method return values is fairly simple to understand. In synchronous RPC, the client stub method only returns to its caller when the remote method has finished. The return value of the remote method is also reported to the caller at that point. However, as the reason for making calls asynchronous is to enable the caller process to continue its execution during a call, an asynchronous client stub procedure must return immediately after sending the call packet. Hence, it can not wait to receive the reply packet from the service process.

To communicate the result of an asynchronous call to the caller, other means than the stub function's return value must be used. The stub immediately returning a *promise object* [LS88] is one such mechanism. A promise object is a token which the caller can later use when it requires the result to continue its execution. The token can be used to wait for the call to finish, and once finished, extract the return value. This is effectively the synchronous RPC stub method split in two parts: the first merely starts the call, and the second waits for it to complete and gives the result. Figure 3.8 illustrates RPC with promise objects. Note how the client can only wait for one promise to be fulfilled at a time. The *future* mechanism [WFN90] is similar to promises, but additionally allows waiting for multiple pending results collectively, so that they can be processed as soon as they are available. In Section 5.1, we will present a refined object-oriented pattern that utilizes implicit invocation to deliver asynchronous results.

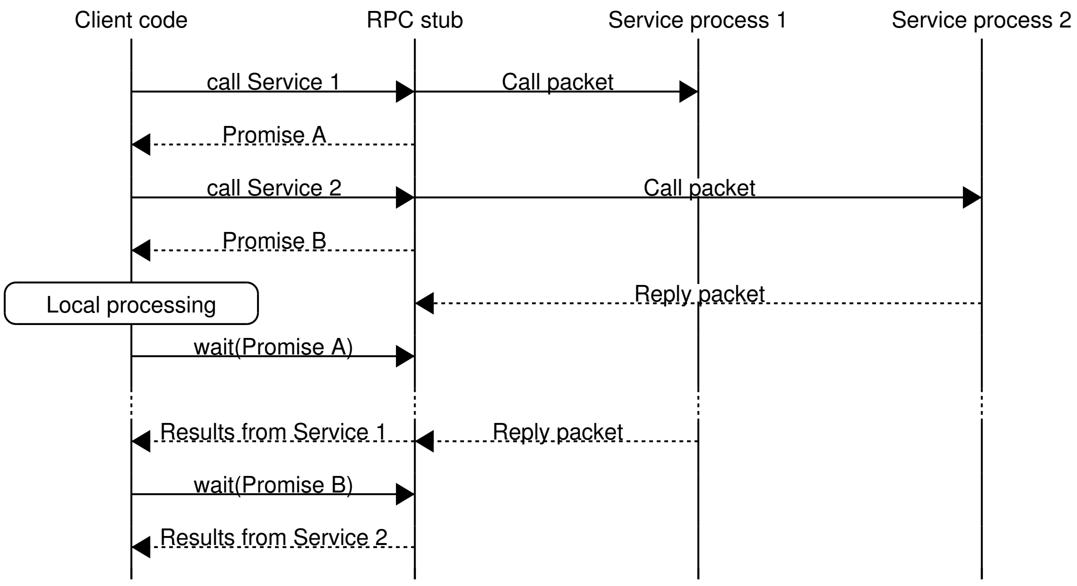


Figure 3.8: RPC with promise objects

3.2 Proxies

In Section 2.1, we described how data abstraction makes it possible to change the internal representation of objects without affecting clients that access them. For a monolithic application, the benefit of this might be restricted to not needing to re-compile the client modules when a change is made. In other words, merely time taken for compilation after a change is conserved. However, if the application is distributed, it is hard to arrange for a re-compile or re-link to occur as needed, e.g. when a service binary has been upgraded, so this alone is a significant benefit [Sha86]. Of course, if there are incompatible changes to an object interface, e.g. some methods are completely removed, or their signature has been changed, even a re-compile does not suffice for either monolithic or distributed applications. The client source code must instead be manually "ported" to the new interface in both of these cases.

The proxy pattern of Shapiro [Sha86] promotes object-oriented data abstraction and encapsulation for inter-process communication. A *proxy object* lives in a client's address space and represents a certain service object, its *principal*, or a part thereof. This is illustrated in Figure 3.9. The method of communicating with the service is considered an internal implementation detail of the proxy. Thus, it is hidden from the client. An unique feature of the proxy pattern, as compared to remote procedure calls, is that instead of a collection of procedures in a flat name space, encapsulated objects which bundle operations together with state are exported.

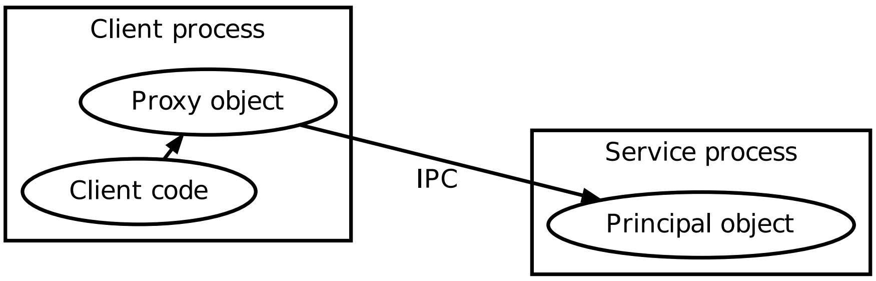


Figure 3.9: Shapiro's proxy pattern

In Shapiro's work, it is assumed that there is operating system level support for migration of objects between processes. Using this mechanism, the entire implementation of proxy objects is downloaded from their corresponding services. Additionally, there is a security framework that restricts inter-process access to service objects to just proxy code. These two properties mean that the service can predict the fashion in which it is accessed, and thus it does not have to take arbitrary special cases into consideration. In Section 3.4.3, we will discuss how the bus daemon process in the D-Bus system can provide similar guarantees, although no such trustworthy object download mechanism is employed.

In addition to not requiring specialized operating system facilities, there are interoperability benefits to not utilizing object migration. This is because the fact that proxy code is downloaded from the service implies that the service must be able to supply code that is executable in and callable from the client. These requirements inhibit cross-architecture and cross-language communication, respectively. For the same reason, inlining (Section 2.1) of remote methods is also hard to implement in a distributed context without restricting callers and callees to a single language and processor architecture.

Just like their principals, proxies are full-blown objects with internal state [Sha86]. Thus, they can be used for more purposes than simple remote procedure invocation facilities. For example, a proxy can store a *local copy* of service object state that is relevant for method preconditions (Section 2.5). Although in systems where services do not supply the proxy code, they have to guard against all kinds of incorrect access, a proxy with a state copy can prevent calls which would fail precondition checks, as an *optimization*.

Additionally, the state copy can be used to run some operations locally without accessing the service [Sha86]. The runnable operations are those with no mutating side effects on the service object, i.e. observation operations ("getters"). Figure 3.10 depicts possible operation of a proxy with a local state copy. We will formalize a state mirroring proxy design that uses D-Bus facilities, and will show its usefulness in environments that are restricted to asynchronous invocation, in Chapter 5.

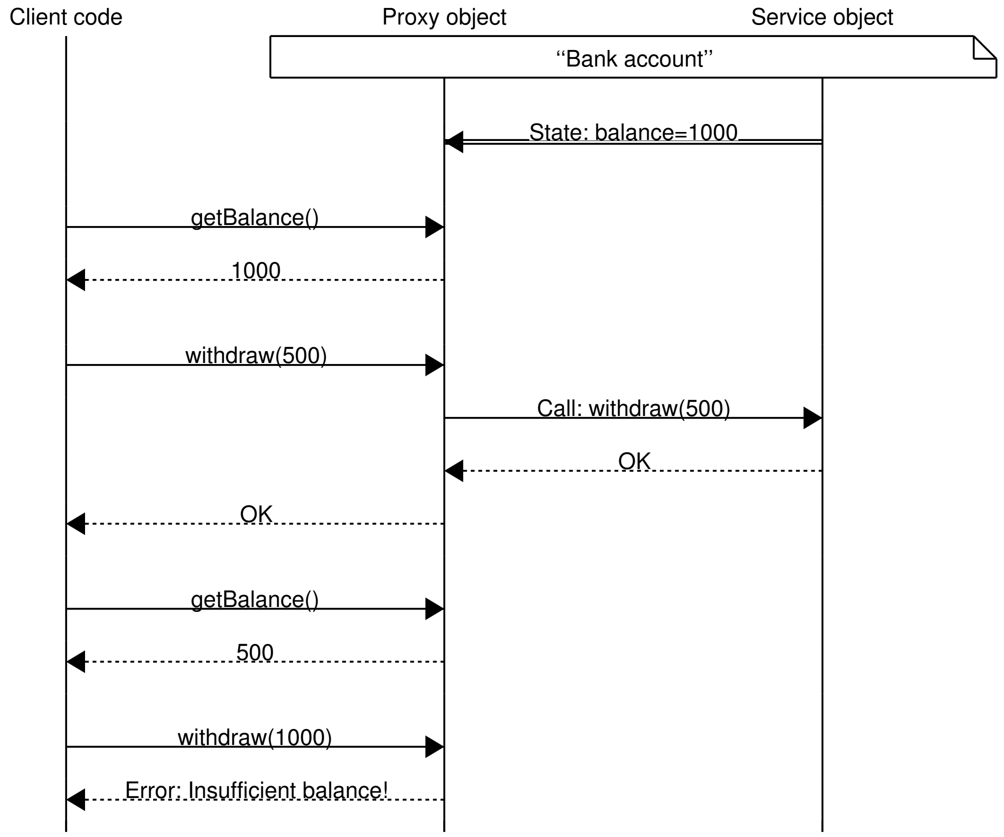


Figure 3.10: Operation of a proxy object with a local state copy

Primitive values and simple *records* composed of them can usually be transferred as byte-for-byte copies when they are used as RPC arguments and return values. A raw copy of an object's internal data can not however be usefully interpreted without the abstract semantics that the object's public inspection operations give to it. Thus, in systems that do not support migration of the implementations of object operations, objects can not be easily passed between processes as arguments or return values. Furthermore, the purpose of some objects is to govern access to a single shared resource. Duplicating such an object would be counterproductive, because then multiple instances would try to access that resource. These challenges seem to be the motivation behind classifying objects as either *service objects* that offer abstract functionality or *data objects*, which are just simple records [OPSS93]. Passing the name/address of an object and constructing a proxy for it in an IPC receiver's address space can be used to provide transparent shared access to non-migratable objects [EGD01]. The use of a proxy to represent an object returned by a service method is illustrated in Figure 3.11.

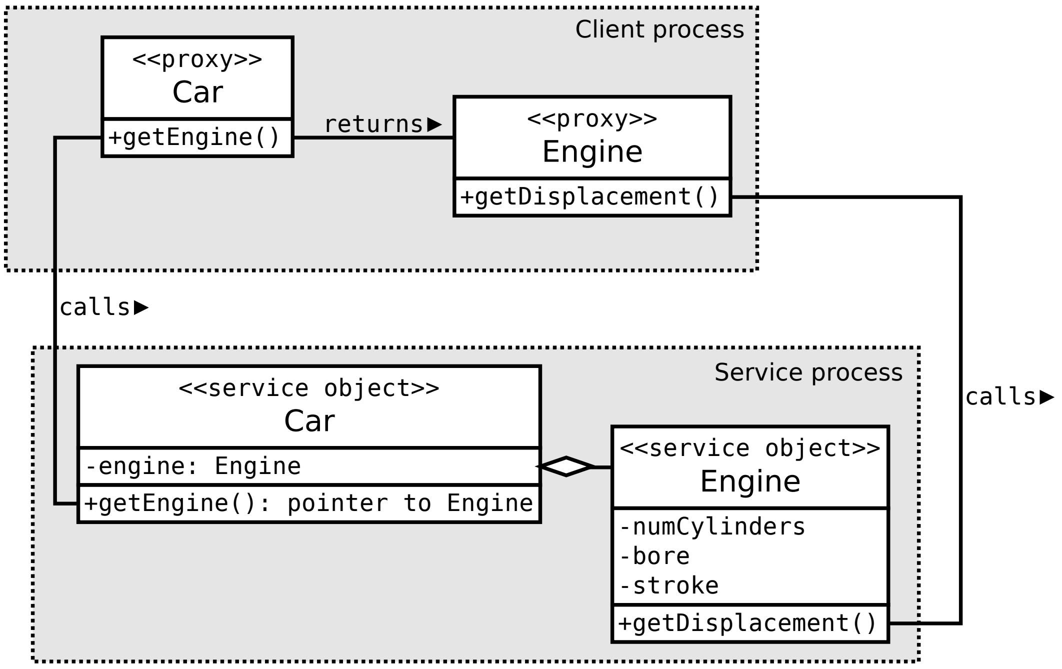


Figure 3.11: Representing an returned service object with a proxy

3.2.1 Proxy-principal communication protocol

In implementations of the proxy pattern that are based on object migration, proxy code can directly access private interfaces of service objects, as the services can trust them to do so responsibly. Hence, a predefined protocol with a restricted set of invokable operations is not needed [Sha86]. However, in other kinds of systems, there must be an adaptor-style object in the service, which is analogous with the RPC service stub. The adaptor implements the service export protocol and binds it to calls to the service object implementation. The relationship between adaptors, proxies and principal objects is shown in Figure 3.12. The role of the adaptor is fulfilled by *message converters* in the distributed knowledge object system of Huang and Duan [HD93]. Both the proxies and the message converters in this system are written by hand, along with the principal objects.

The proxy methods that call the service and the message subtypes in the communication protocol between the proxy and the adaptor both map to the operations that the adaptor invokes on the principal object. Designing these entities separately is therefore

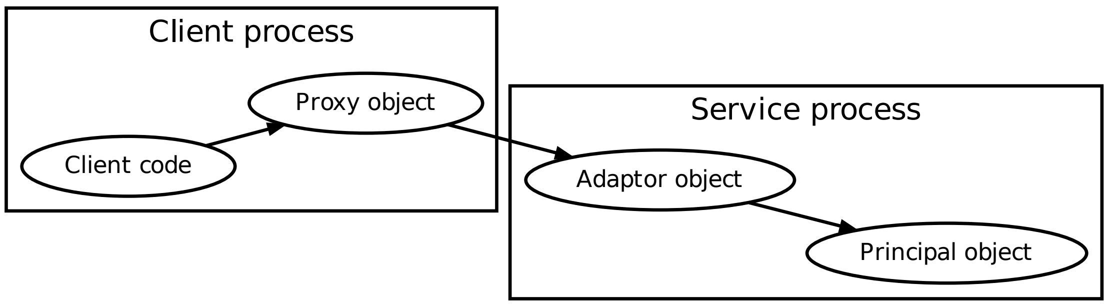


Figure 3.12: Adaptor objects in a proxy-based IPC system

redundant work. The Proxify++ code generator tool in the Choices system inspects C++ class header files to gather information on operations that are marked for export to clients. Then, it generates the corresponding proxies [DSC92]. Thus, the interface is designed only once when writing the service-side implementation. However, in that system, special annotations that detail the semantics of the method interface are needed to guide the encoding of parameters for transfer (*marshalling*).

In the spirit of explicit interface oriented design (Section 2.4), we argue that one should not try to extract IPC interfaces from existing object implementations, especially if they need to be augmented with further metadata. Instead, all of this knowledge should be encoded in a separate interface specification, which will dictate the operations that must be supported by concrete implementations. This is the approach taken in the Java RMI and CORBA systems, for example. In Chapter 6, we will propose an XML-based interface description language (IDL), that can be used to generate proxies for use with the D-Bus messaging system.

Most of our research is concerned with the client side of IPC systems. Hence, we often blur the line between service adaptors and principal objects in our examples, and refer to them together as *service objects*.

3.3 The Publish/Subscribe Paradigm

Remote procedure calls hide the complexity of network communication from inter-process procedural programming, and the proxy pattern translates the benefits of object-oriented data abstraction (encapsulation) to the distributed setting. However, in both paradigms the entity that initiates communication must still specify the receiver using some reference. This can be an explicit address or an abstract name to be translated by a name service. Additionally, the exact operation invoked is always directly specified in RPC, although proxies can transparently remap [DSC92] or even avoid calls depending on the capabilities and state of the remote object, which they can locally mirror [Sha86]. Hence, these mechanisms lack the kind of full decoupling between information *producers* and *consumers* (event announcers and receivers) that is achieved by implicit invocation (Section 2.7).

Event-driven communication can be facilitated in a distributed setting by allowing *publishers* to announce events on some kind of a shared medium, where messages are routed to *subscribers* based on their registered interests [EFGK03]. The interests are registered on the connecting service, and not using application-specific facilities implemented by the publishers. Hence, the publishers are fully decoupled from the identities of the consumers for the events they announce. Additionally, if the rules specifying an interest are based solely on the event itself and not on the identity of the entity which produced it, producers also remain anonymous to their consumers [OPSS93]. The multiplicity of producers and consumers is also flexible: Events from one or more publishers can be routed to an arbitrary number, including zero, of subscribers with a matching interest. The basic structure of the publish/subscribe paradigm [EFGK03] is illustrated in Figure 3.13.

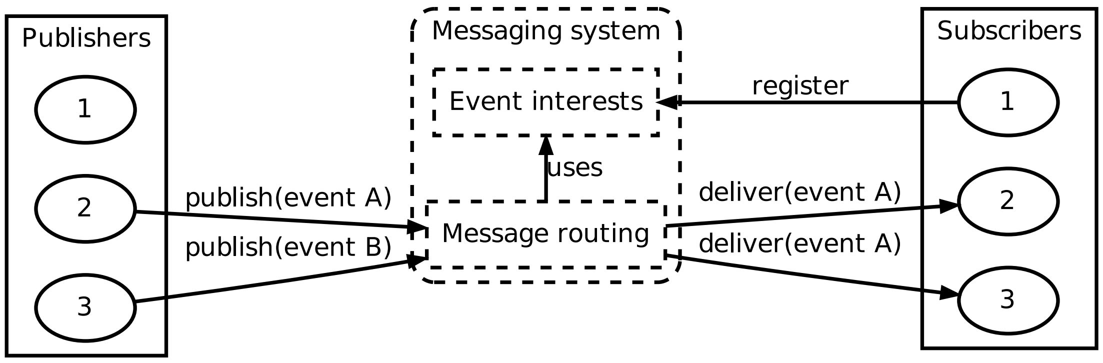


Figure 3.13: The publish/subscribe paradigm

How the interesting events are specified varies in concrete publish/subscribe messaging implementations. In *subject/topic-based* systems, events are labeled with a subject identifier and delivered to all receivers subscribed to that subject. Thus, all events under a certain topic can be thought to be delivered to a *group* of of subscribers. This one-to-many mapping is only altered when subscriptions are made or broken, so it does not need to incur a per-message cost [OPSS93].

In contrast to the above, *content-based* systems allow subscriptions to be based on multiple meta-data attributes of the events, and sometimes even on (parts of) the payload itself. As expressing these kinds of interests is more complex than naming a subject, a *subscription language* needs to be used to specify them. Rules written in this language are interpreted by the messaging system to do message routing. The rules specify ranges of values for certain properties of messages that should make them delivered to the subscriber. Matching messages against the content-based rules implies additional per-message routing work. However, it allows more fine-grained rules than topic-based subscription. Thus, it can help avoid unnecessary message transfer and execution context switches which could result from too broad rules that cause uninteresting events to be delivered to receivers [EFGK03]. Furthermore, when the majority of the event filtering work being done in a central component, common rule subexpressions can be evaluated together as an optimization [EGD01]. The match rules of D-Bus [PCL$^+$12] are also a form of content-based subscriptions. Our D-Bus-based extended proxy construct leverages fine-grained match rules to avoid redundant communication, as will be explained in Section 5.3.

A further refinement to the paradigm is representing events as full-fledged objects—that is, as abstract entities defined by their operations, and belonging to one or more class hierarchies. Implementing this requires an object migration mechanism, and as explained earlier, the use of such mechanisms imposes restrictions on language interoperability. The main motivation for the use of first-class event objects is gaining type safety in the event communication [EGD01]. It is however possible to do type-checking without the events being objects just as it is done for method arguments, if the type signatures of events are specified in an explicit interface description alongside the methods. This is the case in the interface description language we present in Chapter 6.

Although implicit invocation has many uses, certain drawbacks make regular asynchronous RPC calls better suited for some tasks. Hence, it is useful to utilize both of these communication paradigms in a distributed system. Events are by nature fire-and-forget; it is not possible to know whether they were delivered to certain producers or not. Hence, for cases where it must be known that an action was properly undertaken, RPC is more suitable [EFGK03]. A classical example is the trading floor [EGD01]: Stock price updates are inherently multicast-type events, where the most recent update of a given security's price completely supersedes earlier ones. However, the outcome of a single act of purchasing or selling a stock, and that it is attempted exactly once, is very crucial. Those actions should therefore be explicitly invoked operations. A single messaging system can be used to transport operation invocations and replies in addition to events [OPSS93]. This is also reflected in the message types supported by D-Bus (Section 3.4.2).

Concrete publisher and subscriber instances communicating together can be added, removed, altered and replaced freely at runtime due to identity decoupling. A related decoupling dimension is that of *temporal decoupling*: the medium can *store* the event messages in non-volatile memory, and *forward* them only when a consumer is prepared to retrieve them. If this is done, no events are lost even in cases such as temporary network downtime or consumer crashes [EFGK03].

The subscription rule registry, and the store and forward facility, can both be implemented in either a distributed or a centralized manner. In the first style, the rules are stored by each publisher process (though to achieve implementation transparency in the publisher applications, this functionality should be provided by a library). The store and forward functionality is implemented in a shared fashion by using message *queues* in both producers and consumers. Then, either of them can crash or be busy processing something else, but still eventually have the events transferred. Figure 3.14 depicts a distributed publish/subscribe architecture.

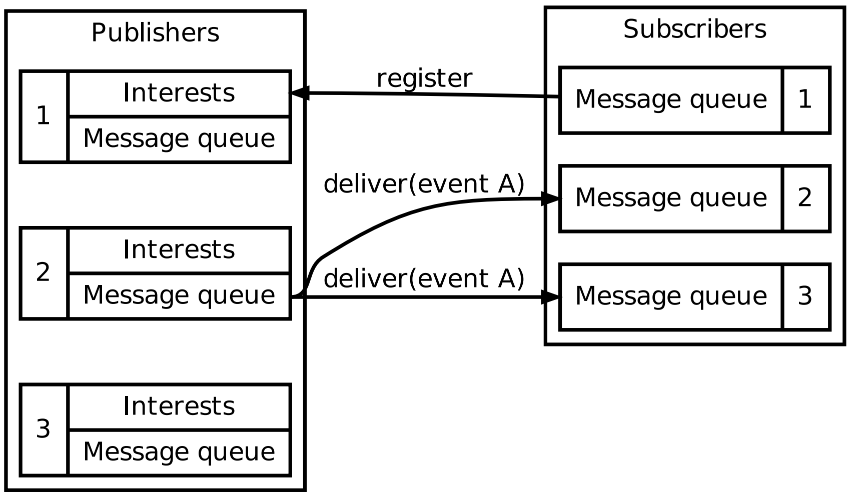


Figure 3.14: Distributed publish/subscribe architecture

In a centralized architecture, both interest registration and guaranteed delivery are implemented by a specific process, or an abstract load-balancing collection of processes [OPSS93]. The parties communicating in the system connect to this central service with point-to-point links [EFGK03]. By connecting relevant publishers and subscribers together, such an entity however provides the illusion of a shared bus topology [OPSS93]. This construction is illustrated in Figure 3.15. In Section 3.4.3, we will describe the subscription-based virtual bus implemented by the bus daemon component of the D-Bus system.

No event store-and-forward facility is included in D-Bus. We assert that in our context of (mostly) node-local systems, communication failures are always resolved in such a short time, that non-volatile event storage would not be useful.

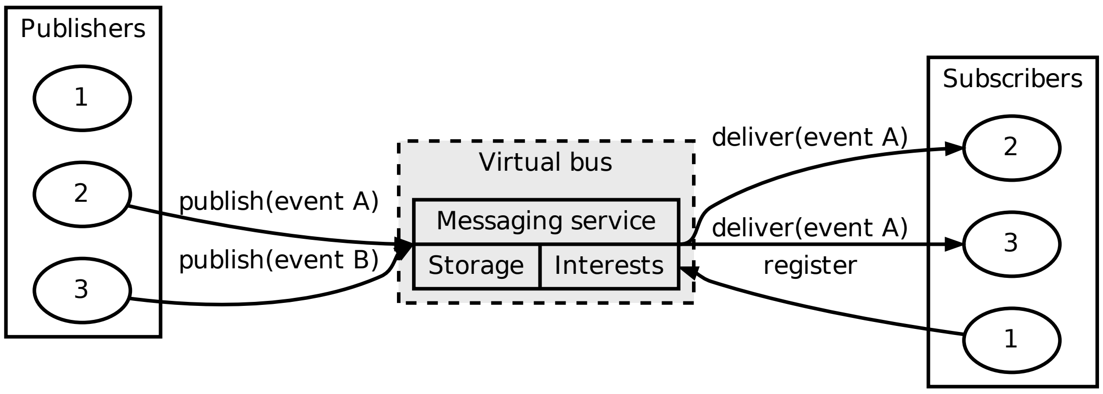


Figure 3.15: Publish/subscribe communication using a central messaging service

3.4 D-Bus

There has been a drive to connect Linux desktop applications together to achieve a more integrated user experience since the 1990s, when the GNOME and KDE desktop environments were born. Sweet et al. have documented [S⁺00] the earliest phases of inter-process communication efforts in the KDE project, which we will now quickly summarize. This is important as the context that led to the eventual creation and adoption of the D-Bus system.

The first IPC mechanism that was widely employed in KDE was built on top of the *X Atoms* mechanism. The Atoms mechanism is a part of the underlying X11 windowing system. It was used in communication between the core visual components of the desktop, such as the window manager and the panel. Although the Atoms mechanism is simple to utilize, it was quickly found to be too restricted and inflexible in terms of what kind of information can be exchanged through it.

In the development of the KOffice office tools suite, it was identified that a more general reusable IPC mechanism was needed. There was an extensive investigation on adopting an implementation of the CORBA system for this purpose. At the time, the GNOME project used the ORBit implementation of CORBA. However, the KDE study found ORBit and other existing CORBA implementations to be unsatisfactory in terms of functionality and completeness. Another problem was their usability with the C++ programming language, which was used for most of KDE's implementation. Also, even the implementation-independent CORBA specification itself was judged to lead to too static software architectures. Simultaneously, the CORBA system was found to be so complex that the learning curve would have been insurmountable for smaller applications. The complexity problem was also evident from characteristics of the prototype, which was built around the MICO CORBA implementation. It suffered from slow compilation and execution, excessive memory usage and unsatisfactory stability.

The DCOP system was then created as a ”simpler CORBA”, to bring the benefits of generic IPC to KDE without the complexity imposed by CORBA. DCOP is based on the X Window System Inter-Client Exchange (ICE) mechanism, which has been a part of all contemporary X Window System releases in the X11R6 series. However, ICE is a very low-level mechanism. For example, its type system is restricted to primitive integer, boolean and character types and homogenous lists of those [SB94]. For this reason, DCOP uses the `QDataStream` facility from the Qt framework for serialization and de-serialization of more complex data [S+00]. While Qt is the basis of KDE, it is not used in the GNOME project, and hence this design choice hampers interoperability with GNOME and other non-Qt systems [Wit05].

D-Bus is a full-featured IPC system. Its type system is designed to be flexible and extensible enough so that it can be used in both GNOME and KDE, without additional incompatible typing and (de-)serialization layers above it [Lov05]. It has since been adopted as the sole high-level IPC mechanism by both of these desktop environments, in which it has replaced CORBA and DCOP, respectively. As such, D-Bus has widespread availability, which has enabled cross-platform projects to depend on it [AFF+09].

In the following sections, we will analyze the features and qualities of the D-Bus system in the context of existing research on object-orientation and inter-process communication. The analysis is based on the canonical specification document, which is published as a `freedesktop.org` standard [PCL+12]. We will specifically focus on the properties which are the most important for the principles and patterns we introduce later. Specifically, we deliberately ignore issues such as the features for peer authentication, message validation, and the lowest level details on the binary representation of messages.

3.4.1 Object model

Services export their functionality over D-Bus in the form of *objects*, which typically, though not necessarily, are backed by concrete native objects in the address space of the service process. The D-Bus object model does not have a concept of classes (as defined in Chapter 2). Instead, the access interface of objects is determined exclusively by the explicit *interfaces* (Section 2.4) that a particular instance implements.

The objects in a single process are identified uniquely by *object paths* of the form `/rooms/Third_floor/301`, that is, a series of elements, which are preceded and separated by slash (`/`) characters. The elements consist of alphanumeric characters and underscores. Object paths are usually mapped to memory addresses of real objects by runtime facilities, but unlike raw memory pointers, they provide an abstract, architecture independent, human-readable and hierarchical identifying system. It is common to leverage these benefits e.g. by exporting some kinds of objects at a fixed well-known path. For example, the Telepathy runtime debugging facility is always present at `/org/freedesktop/Telepathy/debug`. It is also common to give semantics to elements at certain levels of the hierarchy, such as in the paths of Telepathy `Account` objects. These paths include the network protocol and backend implementation used for it as the second- and third-from-last elements, respectively [dTdCDG+12].

The interfaces that D-Bus objects implement are in turn referred to by names like `org.freedesktop.Telepathy.Connection`. Although at the time of writing, there exists no global registry of D-Bus interfaces, an Internet domain name that belongs to the designer or user organization is often included in a reversed form before a number of locally identifying elements, to attain global uniquity for interface names.

Arguably the most important members of interfaces are *methods*, which are functional abstractions, as defined in Section 2.1. Methods are named in a `CamelCase` fashion, and can take an arbitrary number of parameters and also return an arbitrary number of return values. The interface defines the types of these items of data for each method, so that type compatibility can be enforced. The details of the D-Bus type system will be elaborated more in the next section.

Instead of finishing successfully and producing the expected set of return values, a D-Bus method invocation can also result in an *error*. D-Bus errors are analogous to exceptions, which can happen for process-local calls (Section 2.6). Errors have a type, which is a global name with a form similar to that of interface names. For example, the error `org.freedesktop.Telepathy.Error.Channel.InviteOnly` is used in the Telepathy framework to indicate the failure to join a chat room for which an invite is required. Errors can have any kind of data associated with them. However, usually there is just a single string argument that carries a human-readable debugging message, which describes the error in more detail.

Implicit invocation (Section 2.7) is included in D-Bus in the form of *signals*, which are emitted on a particular object instance, but are not targeted towards any specific receiver object. Signals may include arguments. The presence and type of these arguments is defined in the description of the interface which a signal is defined in, as is the case for method parameters. Unlike methods, there is no response to signals, such as a result gathered from the objects listening to it, or even acknowledgement of any process having received it.

As D-Bus interfaces are concerned with how objects are accessed, instead of how they are implemented, they do not include any members with less-than-public accessibility. In a typical encapsulated native object, the data representation would be private to the object—an implementation detail of the data abstraction the object provides. Such

Table 3.1: D-Bus basic types

Name	Code	Description
`BYTE`	`y`	Integer in $(0 \dots 255)$
`BOOLEAN`	`b`	Truth value: 1 (`TRUE`) or 0 (`FALSE`)
`INT16`	`n`	Integer in $(-32768 \dots 32767)$
`UINT16`	`q`	Integer in $(0 \dots 65535)$
`INT32`	`i`	Integer in $\left(-2^{31} \dots 2^{31} - 1\right)$
`UINT32`	`u`	Integer in $\left(0 \dots 2^{32-1}\right)$
`INT64`	`x`	Integer in $\left(-2^{63} \dots 2^{63} - 1\right)$
`UINT64`	`t`	Integer in $\left(0 \dots 2^{64} - 1\right)$
`DOUBLE`	`d`	IEEE 754 double-precision floating point number
`STRING`	`s`	Arbitrary-length string of characters in UTF-8 encoding
`OBJECT_PATH`	`o`	Path to a D-Bus object
`SIGNATURE`	`g`	Type signature of other data

details are not relevant for D-Bus object interfaces; however, they can include publicly accessed attributes in the form of *properties*. These do not need to be backed directly by instance variables of a compatible type, but are usually implemented in services through a getter (inspection) function. If a property is mutable, there is also a corresponding setter (mutator) function. The `Properties` and `ObjectManager` interfaces, and some parts of the design of our state caching proxies, all presented later, are specifically concerned with accessing, transferring and updating these properties of objects.

3.4.2 Message format

Inter-process communication between D-Bus objects takes the physical form of *messages*, which consist of a header and a body. The header specifies how the message should be interpreted, and the body carries the actual data. In particular, the header includes most of the information needed for routing the message to its intended recipient(s).

Type system

All values in D-Bus messages, including both the header and the body, are represented using a well defined type system. The smallest unit of typing is that of the *basic types*. Each basic type has an associated *signature* code, which can represent it by itself or as part of a signature of a more complex type. Table 3.1 lists these data types of D-Bus.

The simplest way to combine primitive types together is concatenation. For example, the signature `iis` would mean that two 32-bit signed integers are followed by a character string. In this case, all three values are separate. A distinct concept is the `STRUCT`, which is represented in signatures by parentheses, such as in `s(yb)`. In this signature, a string value is followed by a single structured value, which is composed of a byte-sized integer and a boolean.

The basic types belong to a larger group, that of *single complete types*. The signature `ii` has two single complete types following each other, while a `STRUCT` groups single complete types together to form a new single complete type. Another way to form new complete types is the `ARRAY` construct. Arrays are represented in signatures by the ‘`a`’ character followed by a single complete type, which determines the type of elements in the resulting homogenous sequence. For example, the signature `ai` specifies a list of integers, and `a(uu)` a list of pairs of 32-bit unsigned integers.

The final primitive for forming complex types is the `DICT`, which is semantically a key-value mapping. Its keys must be unique, but can be of any basic type. The value associated with each key can be of any single complete type. A string-keyed dictionary which has integer triples as values would be written as `a{s(iii)}`.

The `VARIANT` type is represented in signatures by the ‘`v`’ character. A variant can hold a value of any single complete type. To enable the receiver to interpret the value correctly, variants also carry the type signature of the value. Variants also belong to the group of single complete types, so they can be used as array elements and mapping values.

An important property of all single complete types is the ability to *nest* them in variants, structs, arrays and dictionary values. There is a limit to the maximum depth of the nesting, presumably to avoid security and reliability issues caused by stack overflows in message handling code. However, this limit is well beyond any practical uses, at 64 levels in total. This means that constructs such as arrays of arrays of structures are supported. For example, the signature `aa(sii)` would be interpreted as a two-dimensional array of (`STRING`, `INT32`, `INT32`) triples. This nesting capability is what makes the type system of D-Bus truly powerful. The system is able to encode virtually any kind of complex values in a programming environment- and architecture-independent fashion.

We will use D-Bus signatures later on to describe the types of parameters and return values of D-Bus methods, of arguments of signals, and those of properties. As an example, consider the following declaration of a single method. The method takes an array of floating-point values as its only parameter, and produces as its result a single number, the sum of the values in the array.

```
com.example.Calculator.Sum(ad: Numbers) → d: Total
```

Message types

The D-Bus message header has a field for the type of the message, which determines the set of type-specific additional header fields applicable to it, and the interpretation of its body. The first such type is METHOD_CALL, which requests *starting* the invocation of a method. In essence, this is the call packet (Section 3.1) of D-Bus. The method to invoke is identified in these messages by an interface name and the name of the method in that interface. A path is also given to the object instance on which the method should be invoked.

The body of METHOD_CALL messages consists of the parameters passed in to the method, if it requires any. With the description of the interface that contains the method, the signature of the body can be checked against the type signature of the method. All parameters are passed by value—reference arguments are not supported. However, as the object path is a possible value type, mutable reference parameters can be simulated for non-copiable object entities by proxying, as explained earlier in Section 3.2.

The header for all D-Bus messages carries a serial number field, which uniquely identifies messages from a single sender (though not globally amongst all senders). This serial number of method call messages is used to associate with them later METHOD_RETURN messages, which represent successful completion of method execution. The same goes for ERROR messages, which indicate that an D-Bus error/exception occurred. These messages together constitute the reply packets (Section 3.1.1) of D-Bus. As the "packet" terminology never seems to be used in the context of D-Bus, we will exclusively refer to D-Bus call and reply packets as *messages*.

The header of D-Bus reply messages contains a reply_serial field. This is set to the serial number of the call message that they should be interpreted as a response to. Figure 3.16 illustrates the use of this field in practice.

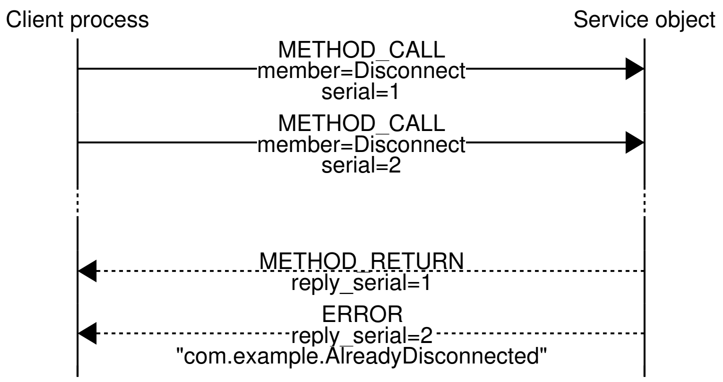


Figure 3.16: The function of the reply_serial field in D-Bus

Method return messages have two purposes. First, they indicate successful completion of a method's execution. If the method produces return values, the message also serves as a carrier for them. This is similar for error messages, except that their contents are usually limited to the D-Bus name of the error. Return messages are only generated and sent by the process that implements the method. However, errors can be generated anywhere in the messaging system, e.g. because the service process has crashed or otherwise became unreachable or unresponsive. Having separate messages for starting method calls and conveying their result enables D-Bus methods to be invoked asynchronously. However, it is still possible for callers to receive results and in general, know the outcome of operations which they invoked. When this is not needed, the `NO_REPLY_EXPECTED` flag can be set in the header of method call messages to prevent a reply from being generated. This can be useful to avoid the cost of sending and delivering the reply message in cases where the result would anyway be ignored by the caller.

Signal emissions are represented by messages with the type `SIGNAL`. Similarly to `METHOD_CALL` messages, their body contains the arguments of the event. However, in their case, the object path does not identify a receiver for the message, but instead the object from which the signal *originated*.

Signal messages are delivered to objects in listening processes primarily based on *match rules*, which are explained in the next section, amongst other subjects related to the D-Bus bus daemon process. However, before that, we will close our treatise of the format of D-Bus messages with a few words on their representation for transfer over communication channels.

Wire representation

Data in textual formats like XML can be easily transferred between systems using network protocols that are intended for transporting human readable text. However, typical processors expect an architecture-specific binary representation for values that are used for arithmetic in executable code. Textual data needs to be converted to this native binary format for processing, which can be expensive. A common difference in the native representations of different processor architectures is whether the most or least significant bits of a multi-byte value should be stored first in memory. This is called *endianness*.

D-Bus uses a binary format to encode messages for transfer. Thus, conversions between a textual format for communication and a binary format for processing are not needed. However, D-Bus specifies binary layout details such as alignment and padding for all kinds of values that are supported by its type system. Additionally, the message header contains a flag indicating the endianness of data in messages, which a receiver can use to convert the values to its native endianness if there is a mismatch. These facts put together mean that although messages are transported in a binary format, this format is architecture independent. Hence, D-Bus can be used for arbitrary cross-machine communication as well. The ”receiver converts” paradigm means that endian conversions are avoided in local-node usage, which improves performance.

3.4.3 The bus daemon

In Section 3.3, we touched the subject of centralized event queueing and routing services in the context of publish/subscribe systems. Publishers and subscribers connect to the service by point-to-point links. However, because the service routes messages between these links, it is possible for any subscriber to consume events from any publisher. Thus, a logical *bus topology* is formed between the publishers and subscribers.

The communication primitives presented in the previous subsections can be used over peer-to-peer links between two D-Bus applications. However, more typical is to communicate through the *bus daemon*, which delivers messages in a bus-like fashion, much like the publish/subscribe central routing services. The structure of this case is illustrated in Figure 3.17.

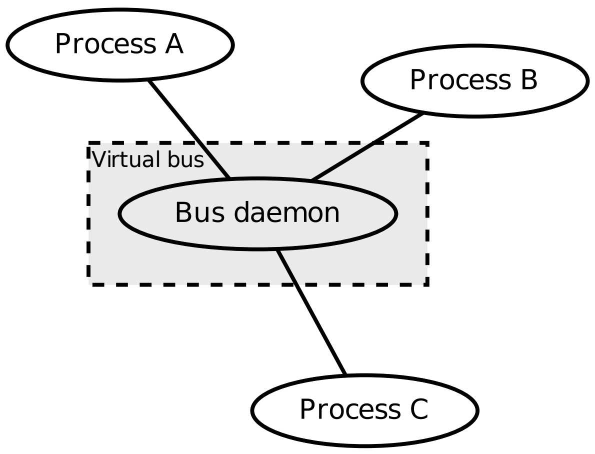


Figure 3.17: D-Bus bus topology

The `DESTINATION` field can be set in D-Bus messages to have the bus daemon forward the message to a particular recipient. This form of *unicast* messaging is always employed for method calls to services on the bus. The bus daemon automatically fills in the complementary `SENDER` field based on the process that sent the message. Services use the value of this field as the destination when building method return and error messages, which result from the method calls.

Every connection to the bus daemon is assigned an *unique name*, such as `:1.234`. These names are distinguished by an initial '`:`' character and are never reused on a bus.

A process can request additional names for itself by the following method implemented by the bus daemon on the *bus service interface*, `org.freedesktop.DBus`:

```
RequestName(s: Name, u: Flags) → u: Disposition
```

The `Name` parameter specifies the name to request. The bus name format is similar to the format for D-Bus interface names, e.g. `com.example.Service`. The flags and details on the disposition return value are only relevant when multiple processes try to claim the same bus name. For our purposes, it suffices to know that with certain flags, the operation only succeeds if the process obtains the name, and fails otherwise.

The destination field of a message can be filled with either a unique name or any other bus name successfully claimed by a service. The bus daemon routes unicast messages according to all the mappings established with `RequestName`. The effect of sending a message to a non-unique name is the same as setting the destination to the unique name of the process which owns that name. These mappings can be broken by their owner by calling the following method of the bus service interface:

```
ReleaseName(s: Name) → u: Result
```

When a name is successfully acquired or released by a process, the bus daemon emits the `NameOwnerChanged` signal on the same interface. Other processes can listen to this signal to be notified of services appearing and exiting. To specify which names are interesting, D-Bus interface specifications often define certain *well-known names* or patterns for them, for implementations of particular functionality. For example, Telepathy protocol backends register names like

```
org.freedesktop.Telepathy.ConnectionManager.gabble
```

where the last component is the name of the executable [dTdCDG+12].

Signal messages can specify a destination as well, to make them routed in a unicast fashion. However, usually this is left out and they are delivered to interested receivers according to *match rules*. The match rules are specified as strings of the form `key1='val1',key2='val2',...` such as

```
interface='org.freedesktop.MediaPlayer',member='StatusChange'
```

Table 3.2 partially describes the semantics of this subscription language. The expressiveness of match rules makes D-Bus a content-based publish/subscribe system. However, there are some restrictions on how content can be matched, as detailed in the table, so it is not a fully general one. Match rules can be added and removed at runtime, so D-Bus supports dynamically altering the connections between event announcers and listeners (Section 2.7). This is done by calling the following two methods of the bus service interface:

```
AddMatch(s: Rule) → nothing
RemoveMatch(s: Rule) → nothing
```

Table 3.2: Some semantics of D-Bus match rules

Key	Example value	Matches on
`type`	`signal`	The type of the message.
`interface`	`org.freedesktop.DBus`	The name of the interface the method or signal the message pertains to is a member of.
`member`	`RequestName`	The name of the method or signal (member of the interface).
`path`	`/rooms/301`	The path to the object to invoke a method on, or which has emitted a signal.
`argN`	"John Doe"	The value of the `N`th argument of the (method call or signal) message, starting from `arg0` for the first argument. Restricted to string arguments.
`arg0namespace`	`com.example`	The initial components of the first argument, which has to be a string formatted like a D-Bus interface name or bus name. The example would match `com.example.EchoService` and `com.example.MusicPlayer`, but not `com.examplesoftwarecompany.Editor`. This is useful for watching a family of names by listening to the `NameOwnerChanged` signal, as the bus name for which the owner changed is the first argument in that signal. However, there are no corresponding match rule keys for arguments other than the first one.
`path_namespace`	`/rooms`	Similarly, the initial components of the path of the object which the message pertains to. The example would match `/rooms/301` and `/rooms/415` etc. but not `/roomservice`.

If a method call is directed to a non-unique bus name that no process currently owns, the bus daemon will attempt to start the service that should provide it. The mapping of bus names to executables that will register them are stored in *service description files*. If the name is found from those files, the daemon will start the service and wait until it has connected to the bus and requested the name. This is called *service activation*. The method call message that caused the service to be activated will be delivered afterwards.

Access control

Any number of D-Bus bus daemon processes can run on a single machine, to create multiple virtual buses. However, there is usually one machine-wide *system bus* for system services, shared between all users, and one bus for each logged in user session (*session bus*). The session bus is typically configured to allow all traffic between connected processes, but the system bus restricts access to destructive actions and confidential information. However, the session buses only allow processes run by their owner user to connect.

To enforce the differences between system, session and other kinds of buses, the bus daemon has a configurable *security policy*. The security policy can, for example, be used to only allow particular users and executables to claim a particular well-known bus name, to prevent spoofing. Also, access to methods and other service properties can be restricted based on user credentials. These restrictions can be used to prevent malicious use of services. In this way, the bus daemon can provide some security guarantees, which would be implicit in a system based on runtime proxy object migration, as we detailed in Section 3.2.

3.4.4 The Properties interface

In addition to the bus service interface implemented by the bus daemon, the D-Bus specification defines a few other standard interfaces. These interfaces contain generic functionality, which is intended to be useful for many kinds of applications.

In Section 3.4.1 we mentioned that in addition to methods and signals, D-Bus interfaces may include properties. There is a standard interface, which objects can implement to expose access to properties on their other interfaces. This interface, `org.freedesktop.Properties`, is presented in Listing 3.1.

Suppose an object implements the standard `Properties` interface, and additionally the `Counter` interface presented in Listing 3.2. The semantics of the `Counter` interface should be self-explanatory. A call of `Get("com.example.Counter", "CurrentValue")` could be used to discover the current value of the counter. The value will be packed

```
Get(s: Interface, s: Property) → v: Value

Set(s: Interface, s: Property, v: New_Value) → nothing

GetAll(s: Interface) → a{sv}: Properties

PropertiesChanged(s: Interface, a{sv}: Changed,
      as: Invalidated) [SIGNAL]
```

D-Bus interface 3.1: `org.freedesktop.Properties`

in a D-Bus `VARIANT` value, as per the return type of the `Get` method.

```
Increment() → nothing

Reset() → nothing

u: CurrentValue [READ-WRITE PROPERTY]

t: LastReset [READ-ONLY PROPERTY]
```

D-Bus interface 3.2: `com.example.Counter`

The `GetAll` method can be used to download all of the properties on a given interface an object implements. This is functionally equivalent to making separate `Get` calls to discover the value of each property, as long as their values do not change between the calls. However, with `GetAll`, only a single method call round trip is needed to download the entire state of an interface. The call `GetAll("com.example.Counter")` would return both the current value of the counter and a timestamp for when it was last reset. The values are returned in a *string-variant map*, with the property names as the keys.

The method `Set` can be used to set object properties. However, it often does not make sense to allow altering the values of some properties. This is the case for the `LastReset` property, for example, since it is supposed to indicate the point of time when the `Reset` method was most recently successfully called. Thus, it is marked as a read-only property in the interface description, and `Set` calls to assign a new value to it would always produce an error.

The `PropertiesChanged` signal is emitted when the value of one or more properties on an object changes. The first argument will specify the interface which the properties in question belong to. The second argument is a mapping with a similar layout as the return value for `GetAll`. Its contents will be new values of the properties.

The last argument of the `PropertiesChanged` signal is probably the hardest part of the entire `Properties` interface to understand. It would be inefficient to transfer the new values of some properties fully as part of the `Changed` map. The `Invalidated` list contains the names of such properties. The information necessary to form their new value is communicated with other means. For example, objects which represent group conversations in Telepathy have a property called `Members`. This property lists all of the members in the conversation. Some conversations might have thousands of members; for them, sending the full list each time one member joins or leaves would be wasteful. The `MembersChanged` signal is emitted for those events. Its arguments specify the members which were added to and removed from the conversation, along some other information [dTdCDG+12]. These changes can be applied to the previous value of the `Members` property by a receiver to discover to the new value.

3.4.5 The ObjectManager interface

The `Properties` interface can be used to download state from a single object and to keep it up-to-date. Clients are sometimes interested in entire sub-trees of service objects, that is, on a root object, and all its children and further descendants. Listing 3.3 presents the `ObjectManager` interface, which is useful for these cases.

```
GetManagedObjects() → a{oa{sa{sv}}}: Objects_With_State
InterfacesAdded(o: Object, a{sa{sv}}: State) [SIGNAL]
InterfacesRemoved(o: Object, as: Interfaces) [SIGNAL]
```

D-Bus interface 3.3: `org.freedesktop.ObjectManager`

The `GetManagedObjects` method can be called on an object to download the state of all objects in the sub-tree rooted at it. The return value is a nested mapping, where the keys on the first level are paths of the objects. The second level keys are the names of the interfaces each object implements. The innermost mapping contains the current state of each of these interfaces. In this mapping, property names are mapped to their values in an identical fashion to the `GetAll` return value. Figure 3.18 shows an example return value with the state of two objects.

The `InterfacesAdded` signal is emitted when a new object appears in the sub-tree, or an existing object gains one or more new interfaces. The second argument maps the names of the interfaces to their state, like the inner two levels of the return value of `GetManagedObjects`. The `InterfacesAdded` signal is emitted when interfaces are removed from existing objects. The last interface(s) being removed from an object means that the object ceases to exist.

The result of `GetManagedObjects` and the arguments of the `InterfacesAdded` signal convey the initial state of all interfaces on the objects. However, changes to the state are not signaled on this interface. The `PropertiesChanged` signal and other more spe-

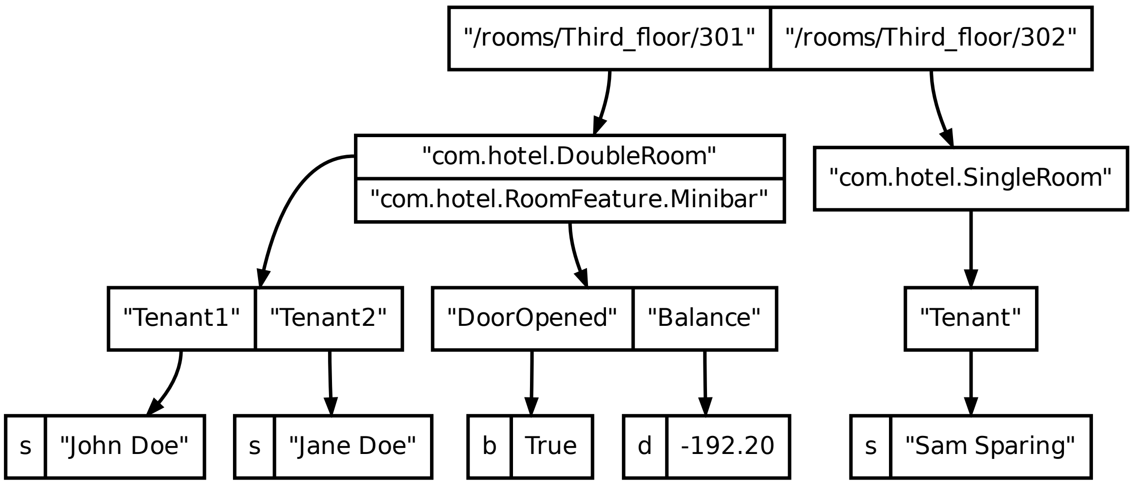


Figure 3.18: Layout of the `GetManagedObjects` return value

cific change notification signals should still be monitored to keep the state synchronized with the service. The idea is that clients using the `ObjectManager` interface will add a match rule which catches all signals emitted in the sub-tree. This will include the change notification signals as well, so they do not need to be subscribed to separately for each object. The `path_namespace` key of match rules can be used to match entire sub-trees, as in the following example:

```
type='signal',name='com.hotel.LondonRiverside',path_namespace='/rooms'
```

Clients which use the `ObjectManager` interface will always receive all information for a given sub-tree of objects. If only some information is actually needed, e.g. a few of the objects, or not from some interfaces of them, the rest is transferred for nothing. Thus, more granular mechanisms should be used to communicate the state of service objects, unless entire sub-trees are really interesting. These finer-grained mechanisms will likely use the `Properties` interface directly, like the state caching proxy pattern that will be presented in Chapter 5.

3.5 Correctness and Scalability Constraints

In the previous sections, we have described the messaging primitives available in D-Bus. It is possible to use this functionality in many different ways to communicate the same information. Here, we analyze practical implications of some commonly emerging messaging patterns. Thus, we can establish constraints on designing messaging such that bad behavior is avoided.

D-Bus communication is done in terms of messages. The messages are delivered on a virtual bus shared by clients. The bus topology is formed by the central daemon, which routes messages between point-to-point connections to each client. Although we use D-Bus as an example case, the conclusions drawn here are applicable to any system which shares these properties.

3.5.1 Message ordering

D-Bus signals can be used to indicate that an event has occurred. A method return message indicates that an operation has finished and describes the operation's outcome. The effect and outcome of methods called on objects depends not only on the parameters passed to them, but also the state the object was in at the time of the call. Some kinds of states might be completely unsuitable for invoking a particular operation—namely, those in which the operation *preconditions* are not fulfilled (c.f. Section 2.5).

Invoking operations on objects that no longer exist does not make sense. Thus, the object being alive and well can be considered an implied precondition of all methods. On D-Bus, one will get an `UnknownObject` or `UnknownMethod` error when calling a method on a nonexistent object [JTM08]. However, this is not always an error on the part of the caller. Some external event might have caused the object to go away just before the call. On the other hand, calling a method on an object which has never existed is always an error. However, in terms of the error code returned, that is indistinguishable from the object having ceased to exist right before the call started processing. It is not feasible for services to provide more information in the error reply, because to do that they would have to remember indefinitely which objects have existed at some point in time. This would consume more and more resources as objects are created and destroyed.

D-Bus objects managed by the `ObjectManager` interface indicate being destroyed through the `InterfacesRemoved` signal, as explained in Section 3.4.5. Thus, it is possible to distinguish the two error cases where the object does not exist by listening to this signal. If the object did exist, but goes away during a method call, there will be an `InterfacesRemoved` signal message before the error reply message. If a client receives and processes the messages in order, it can correctly interpret the error reply as an object having existed, but just now ceased to exist. But if it for some reason processes the reply first, it can not distinguish between the two cases.

For a more concrete example, consider a D-Bus object representing a conversation with multiple participants. Let us assume that current members are able to ”kick” other current members out of the conversations, at which point they cease to be members. This is exactly how e.g. IRC and XMPP multi-user conversations are represented in the Telepathy framework: A `RemoveMembers` method can be called to kick other members. Kicked members get a `MembersChanged` signal, the parameters of which indicate that they were removed from the set of current members due to being kicked by another user [dTdCDG+12]. After being kicked, one is no longer a member and thus can not kick others, and calls to the `RemoveMembers` method would fail. Distinguishing errors resulting from this case from those, where a kick or some other action is attempted even before joining the conversation, requires processing the `MembersChanged` signals and `RemoveMembers` replies in the correct order. Figure 3.19 depicts an ambiguous situation, which can occur if the method reply message is processed first, and other messages reordered after it. The reordered messages are shown in gray.

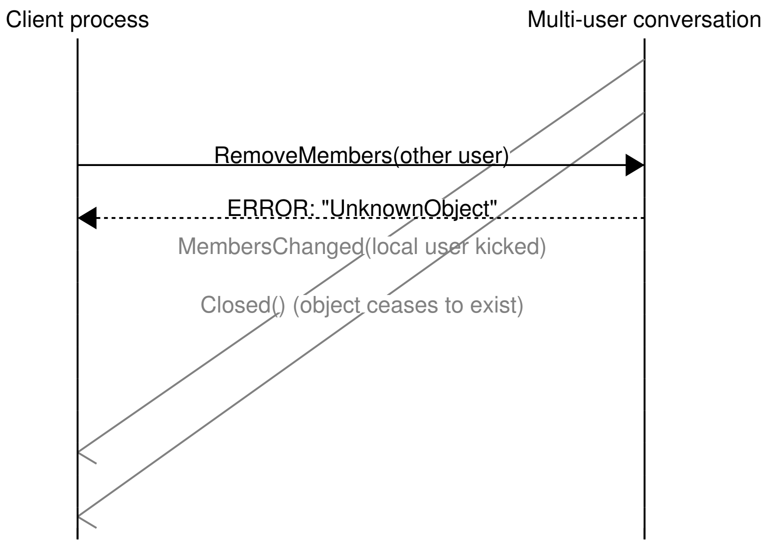

Client process
Multi-user conversation
RemoveMembers(other user)
ERROR: "UnknownObject"
MembersChanged(local user kicked)
Closed() (object ceases to exist)

Figure 3.19: Reordering of a kick event and a method call

One does not even have to mix signals and method calls to see the benefits of consistent message delivery and processing order. Previously, we mentioned that it's not always efficient enough to transfer the new value of object properties fully when they change. Instead, a *delta* from the old value to the new value is transferred as a signal argument. Clearly, if the deltas from successive signals are applied in the wrong order, the end result can be different.

We can conclude that preserving message order is imperative for a messaging system like D-Bus. Luckily, the D-Bus bus daemon guarantees that it delivers messages from a single sender in the same order in which they were sent. However, some client implementation patterns can cause reordering during message processing in the client, as we will see in the next section.

3.5.2 Asynchronous method calls

Previously in Section 3.1.1, we have established that by invoking remote methods asynchronously, a client can run in parallel with the service, or get multiple services working in parallel. This can increase the total system *throughput*, i.e. the amount of work accomplished in a given amount of time.

In addition to throughput, for applications with a graphical user interface (GUI), user-visible latency is also important. The main event loop of GUI applications must be able to dispatch repainting of the interface 30-60 times per second to maintain smooth animation. Users do not see the total number of frames drawn or the average rate of repaints, but they can notice if there is a significant delay between successive frames. Thus, nothing must block the execution of the main loop for significant amounts of time, even occasionally [Pen08].

An application can only guarantee that its event loop runs at a consistent pace, if tasks run by the loop *always* finish in a shorter time than the maximum allowed interval between user interface repaints. Sending a D-Bus method call message can be done in a known-short time, as it's just a matter of sending a small amount of data over the connection to the bus daemon. Similarly, reading a method reply and interpreting it can always be done fairly quickly. However, the client does not have any guarantees about how long the service is going to process the method call until it sends the method reply. Both the method call begin and reply messages can also both end up queued inside the bus daemon for an indefinite amount of time before they even reach their destination. Thus, either initiating a D-Bus method call or processing the reply for one can be considered acceptable within a single iteration of the main loop of an interactive application. However, initiating a method call, waiting for its result and processing it, all between two successive repaints, can lead to a sluggish user experience.

Despite these suboptimal performance characteristics, real-world D-Bus client libraries like `dbus-glib`, `dbus-python` and `QtDBus` provide a mechanism to invoke D-Bus methods and block until their result has been received. We will call this mechanism *pseudo-blocking*, as in Simon McVittie's work on the subject [McV08]. The operation of this mechanism is outlined in Algorithm 3.1.

```
Send method call message
$M \leftarrow$ the SERIAL number of the message
loop
  Wait until there is a complete message from the bus daemon
  if the received message has REPLY_SERIAL=$M$ then
    return the body of the message
  else
    Copy the message to a queue of messages to be processed later
  end if
end loop
```

Algorithm 3.1: D-Bus pseudo-blocking method invocation

In addition to slowing down the client main event loop, pseudo-blocking calls have the potential to completely halt it for a longer period of time. This will occur if the service process has locked up. In this case, the pseudo-blocking algorithm will loop until a timeout is reached [McV08]. In this way, the effects of programming errors causing deadlocks can propagate from services onwards to clients.

The pseudo-blocking algorithm has to push aside D-Bus messages other than the one it is waiting for. Typically, these messages would be processed later, when the application resumes running its main event loop. However, this means that these messages will only be processed after the method return message [McV08]. As demonstrated in the previous section, this can cause the return value to be misinterpreted.

Pseudo-blocking calls also have the potential to lock up two otherwise well-behaving processes, if they happen to call each other at the same time. Let us call these processes A and B. If A uses the pseudo-blocking algorithm for making the call, it will not process incoming messages, such as method calls from B, until it has received the method reply it is expecting. But if B is doing the same, it will not process the method call A sent, and hence will not reply to it. The D-Bus message exchange for this scenario is presented in Figure 3.20. The two processes are in a cycle waiting for a reply from each other, which they will not send because they are reordering incoming method calls after the reply they are expecting. This puts them in a deadlock. We will call this potential cause of deadlocks a *wait-reorder cycle*. Note that this kind of symmetric method calling will occur e.g. if A is using B as a delegatee, and the method call from B is a query for additional information required to complete the operation delegated to it (c.f. Section 2.3 on delegation).

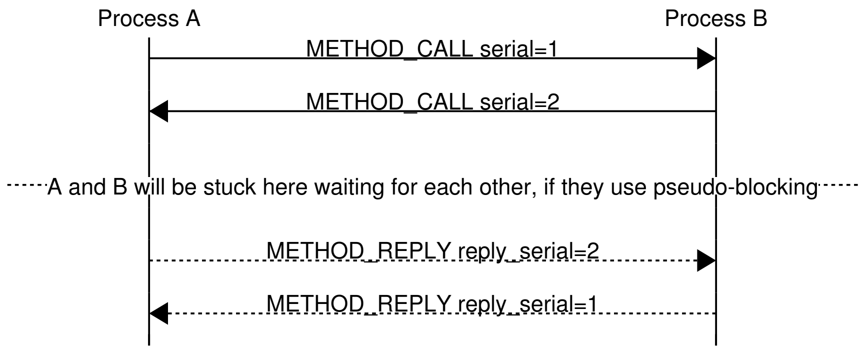

Process A
Process B
METHOD_CALL serial=1
METHOD_CALL serial=2
A and B will be stuck here waiting for each other, if they use pseudo-blocking
METHOD_REPLY reply_serial=2
METHOD_REPLY reply_serial=1

Figure 3.20: Simultaneous method call between two processes

It is possible to break the wait-reorder cycle by making either A or B call the other end asynchronously. If A just starts the method call and then resumes running its main event loop, it is able to process other events, such as incoming method calls from B. The reply to the method will be just another event handled by A's main loop. It is not easy to predict when implementing A, if asynchronous behavior will be needed. It is only safe to make a pseudo-blocking method call to B, if it is known that B will never make a similar call back to A at the same time. In theory, if B is just a D-Bus service, and never makes outgoing method calls, it can not participate in a wait-reorder cycle. However, the implementation of B may change over time. If B has a plug-in system, a plug-in might end up making pseudo-blocking method calls, although the core B application itself would not. If the plug-ins run in the parent application's main loop, they can stall the parent [McV08]. We have seen deadlocks caused by plugins that make pseudo-blocking calls in the development of both One Laptop per Child and Nokia mobile device operating systems.

Changing pseudo-blocking code to be fully asynchronous is not trivial. It requires rewriting almost from scratch the parts of the application that invoke remote methods using D-Bus [Pen08]. This is because with pseudo-blocking calls, the result of the remote method can be utilized right after the call, e.g. by showing it to the user. This is helpful, because the context that led to invoking the remote method is still reachable, and it can be used to decide what the result should be used for. Algorithm 3.2 presents the basic structure of code that makes a single pseudo-blocking method call and utilizes its result. Algorithms 3.3 and 3.4 show similar asynchronous code for comparison. Notably, the algorithm needs to be split to a part which starts the call and to a part which finishes it and utilizes the result. Context needs to be explicitly transferred between the two halves of the algorithm. Because of this significant difficulty of changing pseudo-blocking calls to asynchronous ones, we argue that all code should be written to be fully asynchronous from the beginning.

```
Do local processing
$R \leftarrow$ Result of pseudo-blocking D-Bus method call to B
Use the result $R$ for more processing
return to main event loop
```

Algorithm 3.2: Code that utilizes pseudo-blocking calls

```
Do local processing
Start an asynchronous D-Bus method call to B
Set Algorithm 3.4 to be invoked when the reply is received
Save local context so that the processing can continue later
return to main event loop
```

Algorithm 3.3: Code that invokes a service asynchronously

Re-entrant pseudo-blocking

Some D-Bus client libraries provide a *re-entrant* variant of the pseudo-blocking mechanism. An example of this is the `QDBus::BlockWithGui` call mode of the `QDBus` library.

```
$R \leftarrow$ Newly received D-Bus method result
Resume saved context for the operation
Use the result $R$ for more processing
**return** to main event loop
```

Algorithm 3.4: Code that utilizes the asynchronously received result

In this variant, instead of pushing messages other than the desired reply to a queue, they are processed like the main loop was running again [McV08]. Hence the name—the main loop is effectively re-entered for the duration of the call. Switching to this mode might seem like an appealing alternative to changing code to be properly asynchronous, because it will provide the method result to the caller immediately, just like pseudo-blocking calls. However, as the messages are processed directly, not through a queue, the message re-ordering and deadlock drawbacks of classic pseudo-blocking are not present.

Let us recall the concept of *class invariant* from Section 2.5. The class invariant is a condition that must always hold in an object of a class *unless a method has currently been invoked on it*. Typically, class methods have restored the invariant by the time they return, but it might not hold at all times during their execution. Now, if an object makes re-entrant blocking calls in one of its methods, it might have to process other method calls on itself as a part of the re-entrant event processing. These method calls expect that the class invariant holds. Thus, even though the original method has not returned to the main loop, it has to have restored the class invariant. This is much less obvious than the need to carry state between the two halves of an asynchronous algorithm, and can be harder or even outright impossible to accomplish. Thus, we consider re-entrant pseudo-blocking behavior to be too unpredictable, and hence something that should be avoided.

3.5.3 Message granularity

For a D-Bus message to get delivered from a process to another one, many steps need to happen. First, the message is encoded to the wire format. Then, it is sent over the connection to the bus daemon. The bus daemon uses the information in the message header to decide which processes to deliver the message to. The message is sent to the receivers through the connections to them. Finally, the receiving process(es) decode the message from the wire format and process it.

It is notable that all messages pass through the bus daemon process on their way from one process to another. This requires a *context switch* to the bus daemon, and further ones to the receiving processes. Additionally, at least in Linux, the network traffic between the processes on the bus and the bus daemon causes a context switch to the operating system kernel, because the kernel transfers the data between the processes. Figure 3.21 depicts the minimum context switches needed to transfer a D-Bus message from its sender to a single receiver in this kind of a system.

The same context switches are needed for messages of any size. Large D-Bus messages do not take significantly more time than small ones to transfer [Cre08]. This is consistent with practical results from older IPC systems (e.g. [BN84]). It can be inferred, that context switch overhead dominates the time required for message transfer, and the time taken to actually send the message payload on the wire has less of an effect.

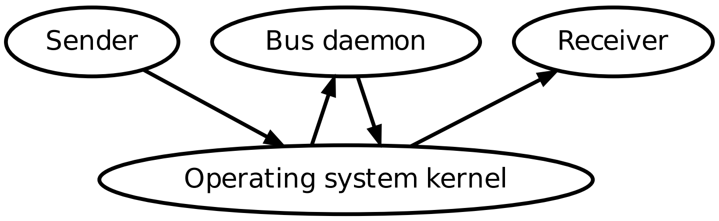


Figure 3.21: Context switches during D-Bus message delivery

By packing more data in a single message, more throughput can be attained, because greater amounts of useful data is transferred between each context switch. It has been found that messages with several dozens of kilobytes of payload give around four times more throughput than messages with 1kB payload, which in turn yield several magnitudes of throughput more than small messages of only a few bytes each. However, most messages in real-world usage of D-Bus fall in the size range from dozens to hundreds of bytes [Tho11]. Thus, it should be made a priority for messaging design to always strive for the largest message size possible. This can be achieved by *batching* related actions and events together. This done e.g. in the initialization of objects following our D-Bus proxy pattern, which will be proposed in Chapter 5.

4 METHODOLOGY

The following chapters introduce principles for the design and implementation of object-oriented systems using the D-Bus system. These principles draw heavily from our experiences on the design of the Telepathy messaging framework [dTdCDG+12]. The Telepathy software will also serve as a source of examples.

The principles we present comprise several practical software *design patterns*. Each pattern will be described formally, as elaborated in the next section.

The impact of the patterns on non-functional qualities of the resulting software will also be studied, where appropriate. Of those, we will especially focus on the effects of the patterns on programming complexity. We will select a few metrics to use for this purpose in Section 4.2.

As established in Section 3.5.3, minimizing the number of D-Bus messages transferred, by increasing the amount of information in a single one, leads to the best throughput. This will be the basis for the discussion of performance properties. A more accurate analysis of the performance qualities would likely be dependent on implementation details, such as whether method calls or signals are processed more efficiently.

4.1 Describing Design Patterns

Design patterns were popularized as a tool for object-oriented software design in a 1994 book by Gamma et al. [GHJV95]. By their definition, a design pattern is a reusable idea that can be applied to solve an abstract design problem recurring in multiple concrete applications. As is the case in the book, the descriptions of our patterns will include the following major elements:

Name A short but descriptive name given to a pattern can be used to easily document and discuss its use in practical designs. A brief description of the *intent* of the pattern expands on the name.

Problem A practical problem is the motivation for the existence of a pattern. In which kinds of cases is the pattern useful? This part identifies the issues which the pattern can help to solve, and thus provides constraints on what is required for it to be applicable.

Solution An abstract outline of how to solve the problem. The *structure* of the solution is shown graphically using Unified Modeling Language version 2 (UML2) class diagrams. Here, the role each *participant* class and object plays is listed, and the *collaborations* that occur between them are described. As the intent is for the solution to be general, these descriptions need to be filled out with application-specific details such as class names to produce concrete designs. We will mention examples of such concrete *known uses* of the patterns to further illustrate how they solve practical problems.

Consequences How will the use of the pattern affect the properties of the resulting design? Here, we will describe the tradeoffs inherent to the use of the pattern, in areas such as performance, complexity and flexibility. The consequences can be crucial when choosing between multiple design alternatives.

As an example of this pattern description format in action, we will now describe the classic Factory pattern from the Gamma et al. book [GHJV95]. A specialized version of this particular pattern will also be presented in Section 5.4.

Example pattern: Abstract Factory

Intent Make it possible to create objects that implement a given interface without specifying their concrete class, so that the concrete classes can be substituted as needed.

Problem A central motivation behind class inheritance is to allow specializations of a given base class to be treated uniformly. For example, a text field user interface element might be a base class, and be specialized for different user interface toolkits. A component that sets the contents of a text field can do so using the generic interface of the base class. However, when creating an object, to attain correctly specialized behavior, a specialized subclass instance needs to be created. If the instance is created directly, the exact subclass will be hard-coded and can not be changed without modifying the creator component.

Participants

- The abstract base class which declares the common interface for objects created by the factory. This is the `TextField` class in the example depicted by Figure 4.1.
- Derived classes which provide specialized behavior for the interface. In the example, these are the `GtkTextField` and `QtTextField` classes.
- Abstract base class for factories that create objects which implement the interface.
- Specialized factory subclasses, which create corresponding derived class instances. The `GtkWidgetFactory` and `QtWidgetFactory` specialize the `WidgetFactory` base class in the example, to create their own types of text field objects.

Collaboration Instead of constructing objects directly, methods of the abstract factory interface are used to create object instances. The factory subclass determines the concrete class of the object produced. A factory can produce just one kind of objects, as in Figure 4.1, or a family of related kinds of objects.

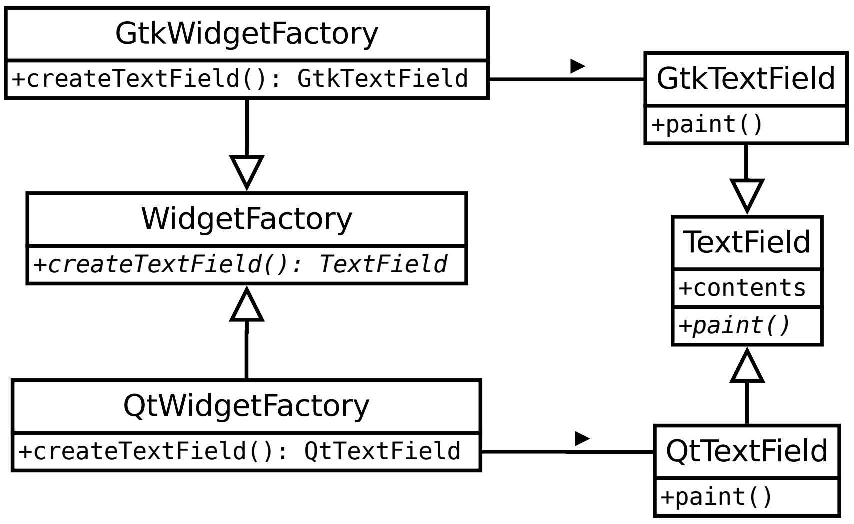


Figure 4.1: Structure of the Abstract Factory pattern

Consequences Code creating objects is decoupled from their concrete classes. This makes it easy to specialize behavior later by just replacing the factory with one that produces objects of a different concrete class. The factory adds a level of indirection to object creation, which can incur a slight performance overhead.

4.2 Predicting Programming Complexity

The complexity of a programming interface affects how easy it is to understand. Understanding an interface is a precursor to using it correctly [GG04]. Hence, to maximize reusability of software components, the complexity of their interfaces should be kept minimal [BA04].

Gill and Grover proposed a metric [GG04] for measuring the complexity of component interfaces. The value of this metric is a weighted sum of three factors:

$$IC = aC_s + bC_c + cCg \ ,$$

where C_s is determined by the *signature complexity* of the interface, C_c by the *interface* *constraints* and C_g by the *interface configurations*. a, b and c are suitable weights, to be determined empirically.

The signature complexity comes from the number of operations and events in the interface:

$$C_s = a_1 n_o + b_1 n_e \ ,$$

where n_o and n_e are the total number of operations and events in the interface. Properties are also considered to be a part of the interface signature, but they are not included in the metric - presumably because accessing them is fairly trivial in non-distributed applications.

Boxall and Araban argue [BA04] that the sheer size of an interface is an unreliable indicator of its understandability and reusability. This is because larger interfaces imply more functionality, and hence possibly being useful for more purposes. Instead, they focus on the number of arguments the interface members have on average. The logic is that procedures with fewer arguments are easier to understand, and thus to reuse. The same reasoning can be extended to events. We can formulate this metric for methods and events commonly as *arguments per member* (APM), given by

$$APM = \frac{n_a}{n_o + n_e} \ ,$$

where n_a is the total number of arguments in all methods and events on the interface, and n_o and n_e are as before.

The constraint factor C_c of Gill and Grover's metric comes from the rules on how the interface can be used. The pre- and postconditions for interface methods (see Section 2.5) increase this value. Contracts that are subjectively harder to fulfill have a larger impact. Another kind of contribution to the constraint factor may come from dependencies between property values, if they can be set to illegal combinations.

The final factor C_g comes from the fact that when a given component is used in multiple kinds of scenarios, it will be configured differently. It might interact with other components in a different fashion, or different parts of its interface may be used. An interface that functions in a similar fashion in all its use scenarios will have a lower configuration complexity.

Boxall and Araban also provide multiple metrics for measuring the *consistency* of an interface. The rationale is that a more consistent interface is easier to learn, because knowledge of a single part of it can be transferred to using other parts. These metrics are mostly concerned with low-level details such as argument naming, and so are not applicable for our pattern-level analysis. However, we will use the basic assumption that consistency lowers complexity in other ways.

Gill and Grover also mention that non-functional qualities such as security, performance and reliability affect reusability. This is because these qualities propagate to an application that uses the component. No formula to produce an aggregate numeric value for these qualities is provided, however. Rather, deficiencies in these areas may rule out reusing a component altogether in a particular application, no matter how simple its interface would be.

These metrics can not be applied in their original form to distributed software. This is because they lack the notion of asynchronous method calls. As explained in Section 3.5.2, placing an asynchronous call necessitates saving the state of the local computation and resuming it in a second method, to be executed once the call completes. Thus, the complexity contributed by an asynchronous method can be thought to be at least twice that of a synchronous one, and possibly more, depending on the difficulty of saving and restoring the state. This means that asynchronous operations are a major contribution to the complexity of remote object access interfaces. This conclusion matches practice—we have discovered asynchronicity to be a major programming challenge.

Interface properties need similar special treatment. The properties forming the state of local objects can be inspected easily by calling a synchronous getter method, usually with no arguments. This kind of method can be considered very simple. However, inspecting the value of a remote object property requires invoking the getter remotely. To avoid the problems described in Section 3.5.2, this call also needs to be asynchronous. Thus, unless special measures are taken, properties also need to be considered fairly complex in the distributed setting.

The exact numerical values of the metrics depend on empirically assigned weights, and will not be comparable between different kinds of components and application areas [GG04]. As we will attempt to come up with generic solution patterns, we will accordingly analyze the trends and asymptotic behavior of the metrics rather than their exact values.

5 DESIGNING D-BUS CLIENT APPLICATIONS

In the previous chapters, we have explored concepts of object-oriented software, the structure of distributed systems in general, and the properties of the D-Bus system in particular. Now, we will describe some mechanisms we have found useful when building object-oriented systems with D-Bus. These mechanisms have been developed in the context of the Telepathy real-time communication framework, but they are general enough to be suitable for other application areas as well. Generic implementations of some of the constructs in fact already exist, for example in the GDBus library [Zeu12].

The bulk of the functionality of a distributed system is implemented in its service components. However, this functionality is only exposed to users by the frontend components. Frontends, especially those with a graphical user interface (GUI), tend to be tied to specific environments more than the backends are. As an example, we have been able to share the protocol backends and account management infrastructure of Telepathy between the GNOME and KDE desktops and various mobile devices. However, most GUI components have been reimplemented for each of these environments. This motivates us to make it as easy as possible for such components (*clients*) to access the services, while still obeying the constraints for messaging design (see Section 3.5). This ensures maximum reusability.

To accommodate convenient access to the services, we have implemented a set of *client libraries* for Telepathy. The Telepathy-Qt library [SMM12] is used by the KDE desktop and Nokia N9 phone frontend software, and the telepathy-glib library [AAB$^+$12] e.g. in the GNOME desktop. These libraries mainly consist of *proxy objects* that mirror interesting service objects in client address space. D-Bus connects the clients and services together. However, the proxy objects hide the actual D-Bus communication behind a friendly programming interface. The following sections detail the design of the proxies and how their communication needs are implemented with D-Bus primitives. Besides proxies, some related helpful constructs will also be described.

The structure of client libraries has to closely follow the design of the underlying D-Bus interfaces, which the service objects implement. However, the only purpose of exporting objects over D-Bus is to allow clients to access them. Thus, we see D-Bus interface design as just a tool to make client libraries as convenient and efficient as possible. Accordingly, the following sections will also touch subjects in interface design.

5.1 Pending Operation Objects

Proxy objects are just representatives of the real objects that reside in service address space. Thus, they need to make D-Bus method calls to the service to implement a lot of their functionality. These calls must be made *asynchronously*, because otherwise the use of the proxies would incur the drawbacks of synchronous calls, as described in Section 3.5.2. Concretely, this means that in proxies, no operation that may potentially call a service with D-Bus can wait for the call to complete before returning to its own caller. This implies that the outcome of the D-Bus operation needs to be reported to the caller via some other mechanism, after the proxy operation has returned.

In Section 3.1.1, we mentioned the concepts of *promise* and *future* objects. These kinds of objects represent the result of an asynchronous computation. When the computation is finished, the result can be extracted from them. The objects support polling to check whether the result is available already, and waiting for it to become available. However, they were designed for performing distributed calculations. In that context, the rationale for making remote operation invocation asynchronous is to make it possible to start multiple operations in parallel, and/or do local processing while a remote calculation is executing. In either case, the caller will have a definite point in its execution, at which it has nothing else to do until it gets the results. At that point, it will wait for the results to become ready and then continue processing with them. This is unlike applications that must remain responsive to user input, have a GUI to animate, or need to serve inter-process method calls themselves. They never have such an opportunity to start waiting for results. On the other hand, periodic polling for the result to be available would be wasteful, and would add latency, up to the polling interval, before the result gets processed.

To avoid both blocking to wait for results and wasteful polling, an *event* (Section 2.7) can be announced to indicate that a previously started operation has finished. The results can be processed in a handler for the event. This can happen as soon as the application has no other work to do, with no added latency.

It is usually necessary to know to *which* invocation of an operation results belong, in order to correctly interpret them. Thus, the operation completion events must include some reference to the particular invocation. One approach is to assign an opaque *operation identifier* to started operations, and to include this value later in the arguments of the corresponding event. This is analogous to low-level D-Bus messages, which include a serial number, and where reply messages specify the serial of the message they correspond to. However, the need to communicate the operation identifier increases the complexity of the event's signature, which shows as an increase in the value of the *arguments per member* metric.

In Telepathy-Qt, running asynchronous operations are represented as objects. These objects are instances of classes derived from the `PendingOperation` base class. No separate operation identifiers are employed—the object identity implicitly determines the corresponding operation invocation. The `PendingOperation` interface includes an event that indicates that the operation has finished, and methods to check whether it finished successfully, or if an error occurred. This information is common to all operations, unlike the type of a result produced for a successful invocation. Subclasses such as `PendingAccount` and `PendingConnection` exist, roughly corresponding to each result type. The subclasses add methods to inspect the value of the result. Using a single base class to represent the common information gives a lower total number of operations and events, which reduces total *signature complexity*. It also makes the way of signaling operation completion *consistent* between all operations.

Sometimes, a client requires multiple operations to finish before it can continue. Often in these cases, the operations do not depend on results from each other, and so they can be started independently. Telepathy-Qt includes the `PendingComposite` class, which makes it easy to wait for a set of concurrently started operations to finish. In a way, this is similar to the *futures* mechanism that was mentioned in Section 3.1.1, but applied to the context of event-driven desktop software.

We will now generalize the scheme of representing asynchronous operations as objects in the Telepathy-Qt library, as a formal design pattern. This will be followed by a similar description of the aggregation mechanism that is implemented by the `PendingComposite` class.

Pattern: Pending Operation Objects

Intent Provide a consistent way to deliver asynchronous operation results once available, without blocking and/or polling.

Problem Interactive applications can never stall their event processing for indefinite amounts of time. Results that take an unbounded amount of time to be produced must therefore be processed asynchronously. Result availability can be announced as an event, but there needs to be a way to associate the results in the event with the original request, in order to correctly interpret them. Adding these events and mechanisms to associate results with requests to a programming interface can increase its complexity if done naively.

Participants

- The base class for all pending operation objects, which includes the common attributes and the event that is used to signal the completion of the operation. This is the `PendingOperation` class in the Telepathy-Qt library, and in the example in Figure 5.1.

- Derived classes for each type of operation result. These make it possible to retrieve the value of the result in an appropriate form. In the example figure, these would be `PendingAccount` and `PendingConnection`.
- Classes that need to have asynchronous operations, such as proxies for D-Bus objects. These create and return the pending operation object instances when an asynchronous operation is started on them. `AccountManager` and `Account` are the classes with this role in the example.

Collaborations Results for asynchronous operation invocations are delivered by having an event announced on the corresponding pending operation object. Client code extracts the values of results using methods declared in the result type specific subclasses. The complete structure of the solution is illustrated in Figure 5.1, with the class names corresponding to those in the Telepathy-Qt library.

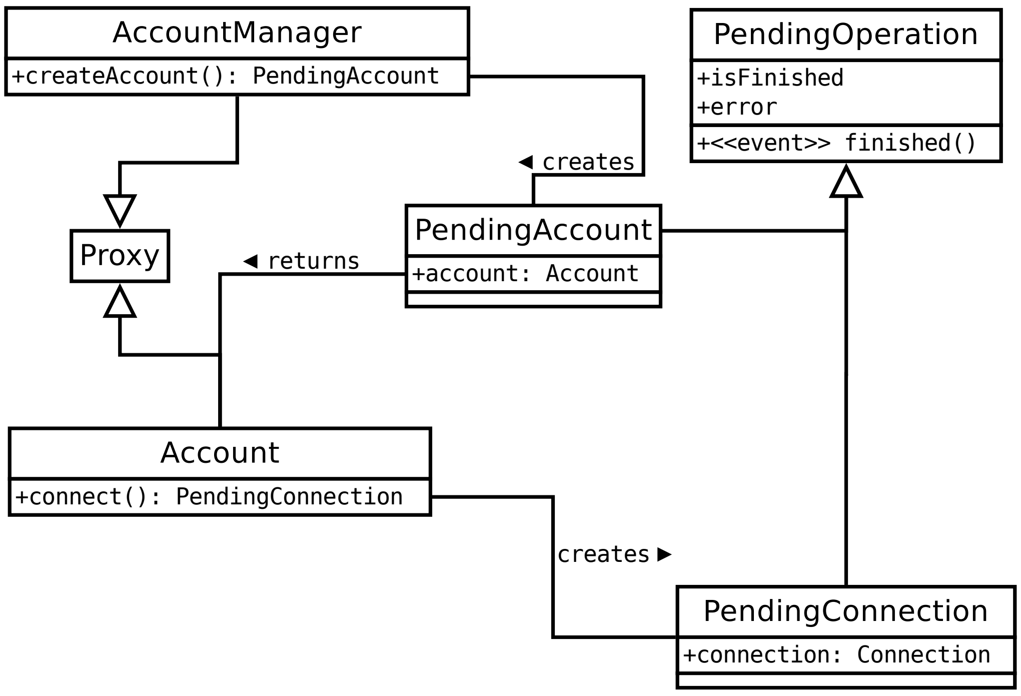


Figure 5.1: Structure of the Pending Operation Objects pattern

Consequences Completion of all asynchronous operations is signaled in a uniform way. This avoids having to re-learn how completion is indicated for each operation, and therefore can make an interface with many asynchronous operations easier to understand. However, creation of the pending operation objects incurs a space and time cost. This cost is usually greater than that of using a simple type, such as an integer, as an operation identifier. If performing the operation itself is very cheap, this might be a significant detriment to performance.

Pattern: Composite Pending Operation

Intent Make it easy to continue processing once a number of independent asynchronous operations have finished.

Problem Resuming processing once an asynchronous operation has finished is by itself complex to achieve. If the completion of multiple operations needs to be considered, this becomes even more difficult. Running the operations as a chain, with one operation finishing causing the next one to be started, and the last one resuming processing, is one way to achieve this. However, that will limit the degree of parallelism by serializing the execution of the operations, which is not strictly necessary unless some operations depend on results from others.

Participants

- Pending operation objects of arbitrary types, which represent the real running tasks. These are instances of `PendingOperation` subclasses in the Figure 5.2 example.
- A composite pending operation container object, which also appears as an pending operation to users. This is the `PendingComposite` class in the example figure, and also in the Telepathy-Qt library.

Collaborations The composite operation object will keep track of individual subtasks finishing. To do this, it holds references to them, as shown in Figure 5.2. Once all of the tasks are done, the composite operation will announce itself to have finished. If users need to inspect the results of individual tasks, they can additionally keep references to them. In any case, they are able to only consider the finish event of the composite operation.

Consequences Waiting for a set of asynchronous operations to finish is as easy as waiting for just one. Essentially, there is just one asynchronous operation instead of multiple ones contributing to software complexity. However, this might make it tempting to overuse composite operations, such that the results of some operations that could be utilized immediately are left waiting for unrelated operations to complete.

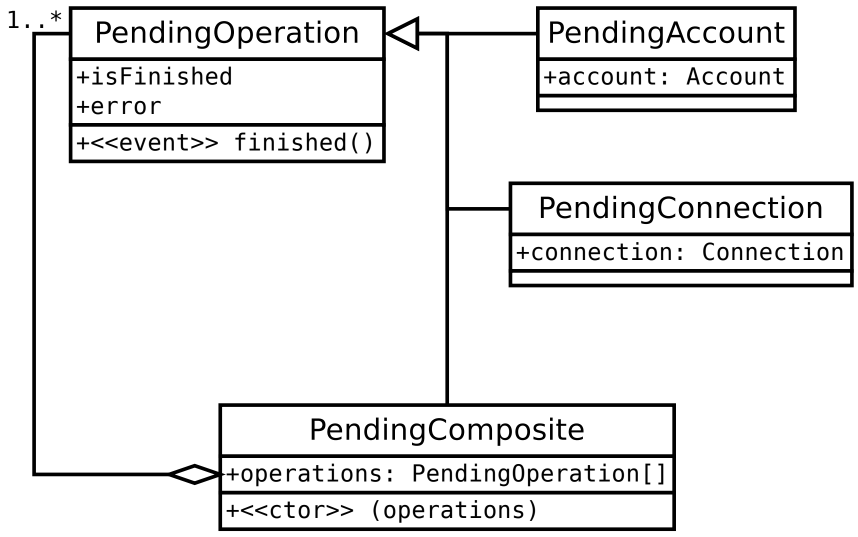


Figure 5.2: Structure of the Composite Pending Operation pattern

5.2 State Caching

The state of local objects can be inspected by calling simple getter functions. However, even a call to a getter function would need to be asynchronous, if made over D-Bus. This makes direct inspection of the state of D-Bus objects very inconvenient. To alleviate this problem, proxies in the Telepathy client libraries store a local copy of the remote object's state. This cached copy of the state can be inspected synchronously without making any D-Bus calls. Thus, state caching could also be considered an optimization, but the main motivation has been to avoid the inconvenience of asynchronous calls to inspect state.

The proxy state cache must be initially populated by querying the state of the remote object using D-Bus calls. In early versions of Telepathy, this was accomplished by calling D-Bus methods such as `GetProtocol` and `GetStatus` on `Connection` objects. These methods resembled typical getter functions to a large extent, with one method for each attribute. However, nowadays the state of Telepathy objects is represented as D-Bus properties (see Section 3.4.4). This makes it possible for proxies to download the state of an entire D-Bus interface at once, using the `GetAll` method [Mad11]. Thus, there are fewer D-Bus messages exchanged in total than if individual getters were called. This leads to less context switches, and thus, better performance (Section 3.5.3).

Regardless of the specific D-Bus methods used to implement the initial state download, it needs to be an asynchronous operation, and proxy state accessors can only be used after it is finished. The net effect is that one asynchronous operation needs to be performed to initialize the cache, but afterwards accessor functions can synchronously return cached values. In proxies for objects with multiple state properties, this reduces the interface signature complexity contributed by asynchronous operations. However, the preconditions of the accessor functions simultaneously become more complex, as they require the state download to have been successfully performed. In Telepathy-Qt proxies, the `becomeReady()` method starts the state download. The telepathy-glib counterpart is the `tp_proxy_prepare_async()` function.

The initial state download only captures the state of the remote object at a certain point of time. If the state changes, the local state copy needs to be updated. Otherwise the values returned by state accessors would get out of date. To keep the state cache up to date, proxies can subscribe to *change notification events* from their remote object. Starting the state download also causes Telepathy proxies to subscribe to the appropriate change notification events. Note that state updates are only propagated to proxy objects when the change notification events reach them. Thus, the service objects and proxies are not always perfectly synchronized. This has seldom been found to be a problem, however, and the same kind of delay applies to naively invoking the getter functions over D-Bus.

The `PropertiesChanged` D-Bus signal is a suitable way to implement change notification for most D-Bus properties. However, the design of most parts of Telepathy predates the addition of this signal to the D-Bus specification. For that reason, custom change notification signals exist in many interfaces, such as `StatusChanged` in the `Connection` interface. As explained before, the `PropertiesChanged` signal is not well suited for announcing changes to properties that have values of a very large size, if the change just affects a small part of the value. Reporting the changes incrementally is more appropriate for such properties. A proxy can be programmed to apply incremental changes to a cached property values to derive the new values.

If D-Bus properties and `PropertiesChanged` change notification are used to represent state of D-Bus objects, it is straightforward to implement state caching. This is because subscribing to `PropertiesChanged` and calling `GetAll` is all that is needed. No interface-specific knowledge is needed on facts such as which getter functions to call and how to interpret custom change notification signals. The `GDBusProxy` class in the GDBus library caches property values in a generic fashion.

When the remote object disappears, values in the state cache lose their meaning. Similarly, operations can no longer be invoked on the object. At this point, Telepathy proxies consider themselves *invalidated*. This might result from normal events such as closing communication sessions (e.g. hanging up a call), or faults such as the backend process having crashed. The exact conditions that cause invalidation depend on the proxy. Typically, they at least watch for `NameOwnerChanged` signals (see Section 3.4.3) to catch the process hosting the remote object disappearing from the bus.

The example in Figure 5.3 illustrates in a simplified form, how the state caching facility of Telepathy-Qt `Connection` proxies works. The state of Telepathy `Connection` objects includes information such as whether the connection is still in the process of establishing a connection to the service, or if that is already done. After the asynchronous initialization operation, client code can query the proxy for the state synchronously. The state queries do not require any D-Bus calls to be made. As the asynchronous initialization operation includes subscribing to change notification signals, the proxy state is kept up-to-date through service state changes. Finally, the proxy object invalidates itself when the connection is disconnected.

Some parts of object state retain the same value for the lifetime of the object. If this is always the case for a certain state property, it does not need change notification. We will call such properties *immutable*. If the value of an immutable property is in some way known previously, it does not need to be rediscovered when initializing the state cache of a proxy, as it can not have changed in the meantime. For example, `Channel` objects in Telepathy represent active communication sessions, such as text chats and video calls. The properties in the main `Channel` interface describe what kind of a session the `Channel` represents—whether it was initiated by the local user or a remote one, whether there is a group of peers or just one, and so on. All of these properties in the `Channel` interface are immutable. When the creation of a new `Channel` is announced, the values of the immutable properties are included in the signal arguments. This makes it possible for Telepathy proxies to skip the `GetAll` call for properties on `Channel` and other similar interfaces. Instead, they can initialize the state cache with the values from the creation signal. This optimization is currently not possible with the generic state caching facility of GDBus proxies.

Next, we will give a formal description of the above proxy state caching mechanism. The example case in this description is a simplified form of the corresponding parts of the Telepathy-Qt library.

Pattern: State Caching Proxy

Intent Allow inspecting the state of remote objects synchronously, without hitting the usual problems associated with synchronous D-Bus calls.

Problem Inspecting the state is a common operation in the use of many kinds of objects. That is needed e.g. to present information from an object to the user, or to decide whether some operation could be invoked on the object. The state of local objects can be inspected by calling simple getter methods. However, when an object is accessed over D-Bus, invoking getter methods is not as simple. Synchronous calls to them could block the

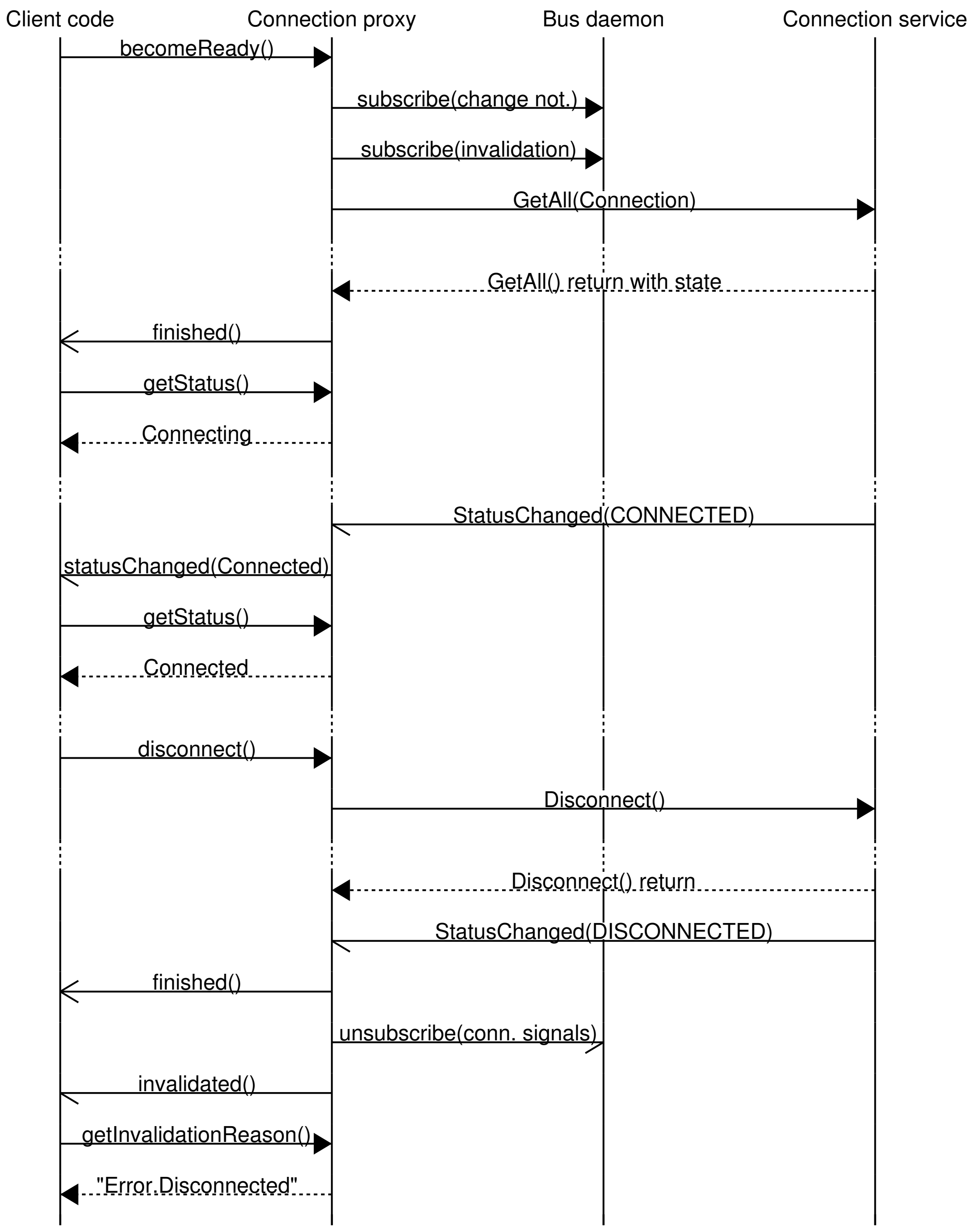


Figure 5.3: Life cycle of a Connection proxy

calling thread for an indefinite amount of time or lead to deadlocks. Making the getters asynchronous increases interface complexity.

Participants

- Service objects which expose their state and changes to it through their D-Bus interfaces. The `Connection` and `Channel` service objects have this role in the Figure 5.4 example. These objects expose their state as D-Bus properties using the `Properties` interface.
- Proxy objects that cache service object state. In the example figure, there is a proxy for each of the `Connection` and `Channel` service objects.
- The bus daemon, which can inform clients that another client has disconnected from the bus.

Collaborations Proxy objects subscribe to change notification and download a copy of service object state in an asynchronous initialization operation. The cached copy of state is kept up to date as changes occur. This is done over D-Bus using the `Properties` interface, as shown in Figure 5.4. Client code can hence inspect the cached state values synchronously. The proxies also monitor when their state copy becomes invalid, as per interface-specific criteria.

Consequences Inspecting the state of remote objects becomes as easy as that of local objects. However, an initial asynchronous operation must be performed before the state can be inspected, which readds some signature complexity. Being subscribed to change notification events causes the client process to be woken up whenever changes occur. If the new value for a state attribute is not needed by a particular client, the wakeup for it is an unnecessary context switch. Such unnecessary context switches delay resuming the execution of processes with useful work to do, and returning to processor sleep states. This can be significant especially for mobile devices, where the use of sleep states affects battery runtime.

5.3 Opt-in to Avoid Unnecessary Wakeups

Let us recall the concept of object *roles* from Section 2.4. The idea is that each kind of a user for an object will use it in different ways. In other words, the object plays a different role for each user. This is no different for D-Bus objects. A part of the rationale for making software distributed was to allow multiple different frontend components to utilize

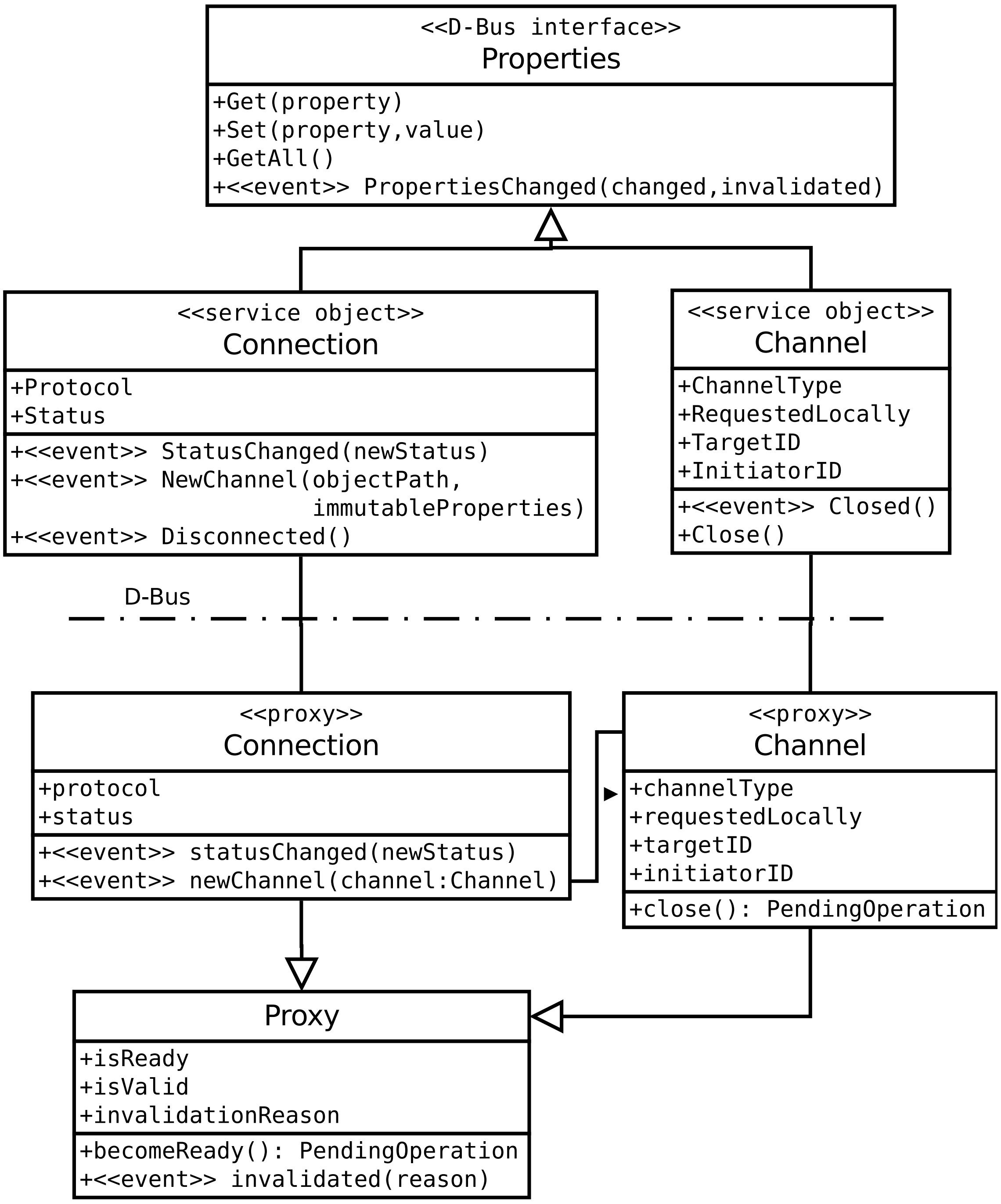


Figure 5.4: Structure of the State Caching Proxy pattern

services from a backend at once. The exact set of functionality employed will naturally be different for each frontend. It follows that if proxies expose all of the functionality of a service object to all clients, some of it will be redundant.

Offering redundant functionality is not a problem in itself. However, because keeping a proxy's state cache up to date necessitates subscribing to change notification events, caching unneeded pieces of state causes unnecessary wakeups. As noted in the previous section, this can be detrimental to performance and power usage.

The harmful effects of proxying unneeded functionality is not limited to wakeups for change notification. D-Bus signals are used for announcing other kinds of events than just state changes. These events are relayed by proxies to the client in a suitable form. If some events are not interesting, the wakeups caused by them will also be unnecessary. Additionally, many proxies in Telepathy client libraries aggregate information from multiple D-Bus interfaces implemented by service objects. This is done to provide a more natural programming interface. However, if the state of a particular interface is not needed at all, the `GetAll` method call to fetch it will be wasted, increasing both D-Bus traffic and proxy initialization latency for no reason.

To solve this, some functionality of Telepathy proxies is conditionally enabled. These individual subsets of functionality are called the *optional features* of the proxies. For example, proxies for text chat objects report the unread messages in the chat. This functionality is likely needed by anything observing a text chat session. However, there is also support to report the active state of the chat—whether the remote user is currently typing, is inactive, and so forth. This information can be instantly displayed to a user. However, it makes little sense to permanently store the chat state in a log, as it is transient by nature. Accordingly, both a user interface (UI) application for text chatting and a daemon that logs conversations to permanent storage would utilize the message buffer, but only the UI application would enable the feature that causes typing notifications to be delivered.

The features can be individually enabled, or multiple features can be activated in a single asynchronous operation. The corresponding signal subscriptions and state download will be performed behind the scenes by the proxy. Note that this depends on the capability to dynamically subscribe to the change notification signals, and thus the dynamic nature of D-Bus match rules.

Features further increase the complexity imposed by the asynchronous initialization of proxies: availability of proxy functionality is no longer just a matter of whether a proxy has been prepared for use, but also whether suitable features have been requested while doing so. We will consider these effects as a part of the following formal pattern description.

Pattern: Optional Proxy Features

Intent Avoid wakeups from unneeded events on proxied remote objects, and initialization work pertaining to those parts of state that are not interesting to a given client.

Problem Service objects tend to have a wide array of related functionality. However, any given user of an object is likely to utilize only a subset of it. Proxying all of the functionality to each user would cause redundant work to be performed.

Participants

- Feature objects, that signify subsets of proxy functionality to enable. The `Chat State` item in Figure 5.5 is a feature object.
- Proxies, on which functionality can be conditionally enabled. In Figure 5.5, some functionality of the `TextChannel` proxy requires the `Chat State` feature to be enabled.

Collaborations Client code requests the necessary features to be enabled when making a proxy ready for use. Subsets of proxy functionality become available, once the operation where they were requested has finished.

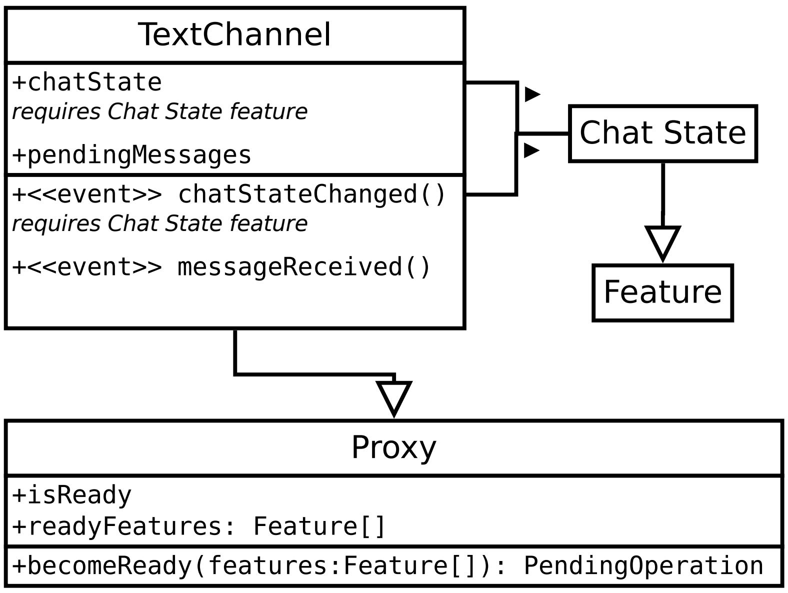

TextChannel
+chatState
requires Chat State feature
+pendingMessages
+<<event>> chatStateChanged()
requires Chat State feature
+<<event>> messageReceived()
Chat State
Feature
Proxy
+isReady
+readyFeatures: Feature[]
+becomeReady(features:Feature[]): PendingOperation

Figure 5.5: Structure of the Optional Proxy Features pattern

Consequences When proxy functionality is carefully divided to features, redundant work to offer unneeded functionality can be avoided. However, the preconditions of the conditionally enabled proxy members become more complex as a result. This can also be seen to increase the configuration complexity factor of Gill and Grover's interface complexity metric (see Section 4.2). This is because client code may need to prepare to use proxy objects with multiple configurations of features enabled. Cautious measures to prevent this might include enabling more features than is actually needed, which defeats the purpose of optional proxy features.

5.4 Asynchronous Proxy Factories

Not all Telepathy proxy instances are created directly by application code. Some of them are created by other proxies and other client library components. For example, Telepathy `Account` objects store information such as credentials, server addresses, etc. that is necessary to connect to a communication service, such as an Instant Messaging (IM) network. A `Connection` object, in turn, represents an active connection to such a service. `Account` proxies create a `Connection` proxy to represent the connection whenever one exists. When disconnecting from the service, the `Connection` object ceases to exist. At this point, the `Account` proxy will also drop the corresponding `Connection` proxy.

For proxies to be useful to application code, their asynchronous initialization operation must be finished. Furthermore, the proxy features that the application uses need to have been requested in the operation. This is no different for proxies created by client library components. Consider how this could be accomplished for the `Connection` proxies created by an `Account` proxy. When an `Account` announces that there is now a connection and a proxy for it, application code could start the asynchronous operation to initialize the proxy. However, that operation might not finish before the connection goes away. A new `Connection` might even appear during the operation. These various possible sequences of events make it rather complicated to handle the events for proxies appearing, becoming ready to use and disappearing.

Instead of giving out unprepared proxy objects, the library components can perform the asynchronous initialization before announcing a proxy to the application. Then, the components can internally handle the complexity of preparing the proxies while the corresponding service object might be replaced or dropped. But which proxy features should they request to be enabled?

The Telepathy-Qt client library includes a set of factory classes that construct proxies. These factories allow overriding the class type of created objects, following the classic Abstract Factory pattern that was presented in Section 4.1. But more importantly, they can be set up by the application to enable desired features on the created proxies. As enabling features is an asynchronous operation, it is necessary for the object creation operation on the factories to be asynchronous as well. This operation is usually completed inside library code, however. By the time a proxy is given to the application, the features that the factory was configured to enable will be ready. Thus, application code only has to deal with the complexity caused by the initialization of those proxies that it itself creates.

The factories can also be configured to perform no setup work at all. This is useful e.g. for an account configuration application: it is concerned only with the parameters of the accounts, and will not care about connections to them. By configuring the connection factory to not make anything ready, no redundant work will be performed to initialize `Connection` proxies.

A side benefit of creating proxy objects through a factory is that the factory can cache the created proxies. This is useful, as multiple components might want to expose a proxy for a given remote object at the same time. In this case, only the first request for a proxy from the factory will lead to the creation of an instance. Succeeding requests will reuse the object created for the first request. Once no references to a proxy remain outside a factory, the proxy will be removed from the cache and destroyed. This is done so that an unused proxy will not keep the application subscribed to events from the corresponding remote object.

We will now formulate this scheme as a design pattern. To keep the description focused on the more novel features of the pattern, the factory is depicted as creating objects of just one kind. However, like in the classic Abstract Factory pattern (Section 4.1), a factory object might be responsible for the creation of objects of a family of related types. In fact, similar functionality in the telepathy-glib library follows this design, with a single factory class creating all of `Account`, `Connection`, and `Channel` proxies.

Pattern: Asynchronous Proxy Factory

Intent Allow specifying what kind of proxies and with what features enabled should be given to application code, so that it does not need to deal with complex asynchronous initialization by itself.

Problem Performing asynchronous initialization of proxy objects is a complex task. Library components could do this setup work before giving proxies they create to application code. However, if the components blindly enable proxy features that the application does not actually need, some setup work may be wasted, and unneeded events may get subscribed to. Also, if multiple components want to proxy the same remote object, they can end up creating and initializing duplicate proxies.

Participants

- A proxy class, which may have specialized subclasses. In the example depicted by Figure 5.6, this is the `Connection` proxy.
- Proxy factory, which can create proxy objects with pre-set features ready. Has an internal cache of proxies to avoid constructing multiple proxies for the same remote object. In the example, the factory constructs just `Connection` proxies.
- Optionally, subclasses of the proxy factory, which construct specialized proxies. The subclasses override a private operation that is called by the base class if a requested proxy is not found in the cache. The greyed out part of the example figure depicts these optional components.
- Components that create proxies, without knowing the exact subclass they end up being, or what features should be prepared on them. In the example, this role is filled by the `Account` and `ChannelObserver` classes.

Collaborations Application code sets up factory instances with the features it requires to be enabled on proxies. Library components request proxies from the factories with an asynchronous operation that finishes once the configured features are ready. The components only announce the proxy to the application when it is ready to use, if it is still meaningful at that point. Application code can also subclass the factory classes to create proxy instances with a custom derived type. These interactions are illustrated in Figure 5.6.

Consequences Application code does not have to perform asynchronous initialization of proxy objects created by library components. This lowers the effective signature complexity contributed by the asynchronous setup operations for these objects. Library components can share proxy instances created for each other, so redundant setup work is not performed or memory wasted for duplicate instances.

5.5 Multiplexed State Synchronization

Normal proxies do at least one `GetAll` call to download the initial state of the corresponding remote object. This is fine if there are relatively few objects to proxy. However, certain kinds of objects can exist in very large numbers. For example, large chat rooms can have hundreds or thousands of member contacts, the details of whom are natural to represent as objects. It would be good for performance if the state of such objects could be downloaded in bigger batches than one contact per method call.

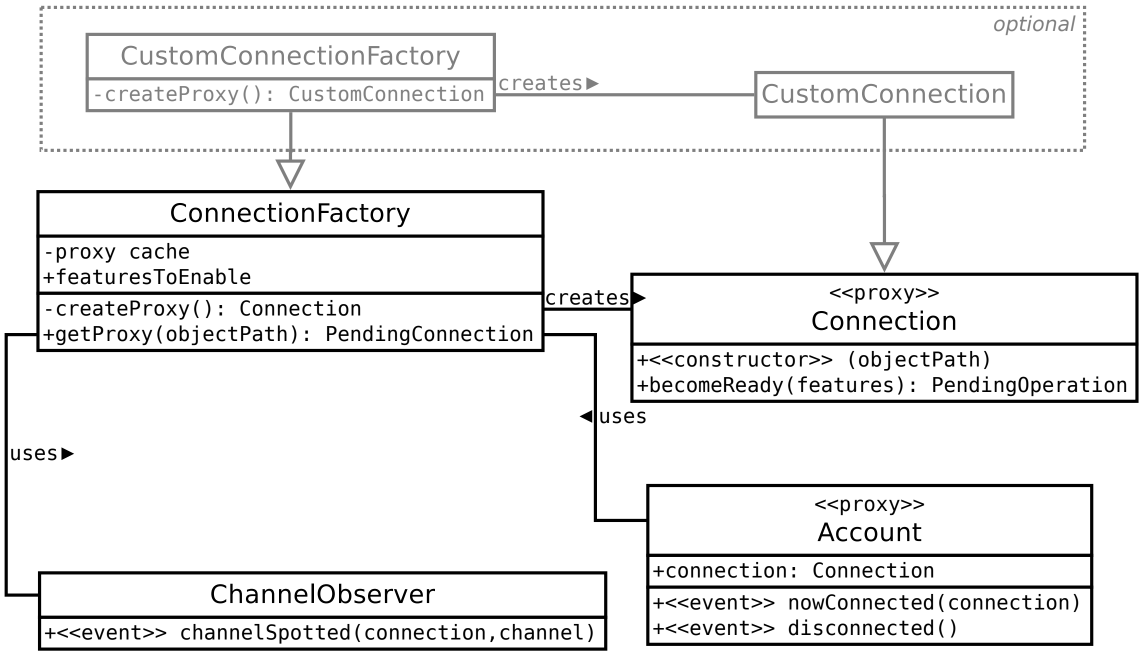

optional
CustomConnectionFactory
-createProxy(): CustomConnection
creates ▶
CustomConnection
ConnectionFactory
-proxy cache
+featuresToEnable
-createProxy(): Connection
+getProxy(objectPath): PendingConnection
creates ▶
<<proxy>>
Connection
+<<constructor>> (objectPath)
+becomeReady(features): PendingOperation
◀ uses
uses ▶
<<proxy>>
Account
+connection: Connection
+<<event>> nowConnected(connection)
+<<event>> disconnected()
ChannelObserver
+<<event>> channelSpotted(connection,channel)

Figure 5.6: Structure of the Asynchronous Proxy Factory pattern

The standard `ObjectManager` mechanism of D-Bus (Section 3.4.5) allows downloading the state of an entire object sub-tree in a single D-Bus call. If the chat member objects were in a tree that is rooted at the chat session object, this mechanism could be used for efficient download of the state of the chat and all its members. However, in Telepathy, contact objects are conceptually children of the connection to the messaging service, not the individual chat sessions. The sessions merely have references to the contacts. This scheme is depicted in Figure 5.7. The rationale behind this approach is that that the same contact objects could simultaneously be members of multiple chats and e.g. be in the permanent address book of the user. It follows that it would only be possible to retrieve the state of all contacts of a connection at a time with `ObjectManager`. This would be wasteful for a process that only shows the contacts in the address book, or those in a particular chat room.

To provide an efficient way to download the state of a given subset of its contacts, Telepathy `Connection` objects implement a D-Bus method called `GetContactAttributes`. The first argument to this method is a list of contact identifiers, which specify the contacts to fetch information for. The identifiers are integer handles which map to underlying protocol addresses for the contacts. While it is efficient to perform set operations such as testing for membership with integers, others [Pen07] discourage referring to objects by anything other than D-Bus object paths in D-Bus interfaces.

The second argument of `GetContactAttributes` is a list of D-Bus interfaces. These

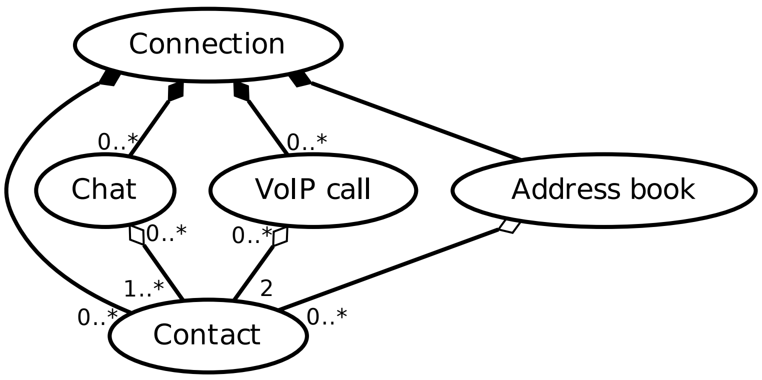


Figure 5.7: Contact ownership in Telepathy

specify which aspects of contact information are interesting to the caller. The return value includes the corresponding state attributes for the requested contacts. In the client libraries, the interfaces are represented as optional features, in a similar manner to those that can be enabled on proxies (Section 5.3).

The client libraries use the `GetContactAttributes` method to initialize local objects representing contacts. These objects are not proxies in the previously established sense, as they do not perform D-Bus traffic to a corresponding service object on their own. Instead, they represent information retrieved by a `Connection` proxy from a `Connection` service object. Thus, the service object conceptually *multiplexes* data from several contact objects on the service, and the proxy *demultiplexes* this data to separate objects. This allows downloading the state of an interesting set of contacts in one batch. However, the contacts are still represented in an intuitive fashion—as separate objects—in client-side code. Besides initial state download, change notification is also multiplexed. We will now attempt to generalize this overall mechanism as a formal design pattern.

Pattern: Multiplexed State Synchronization

Intent Allow downloading the state from several service objects at once, without the batching increasing client application code complexity.

Problem Downloading state individually for a large number of objects imposes D-Bus message header and context switch overhead in the order of the number of the objects. The `ObjectManager` D-Bus interface is a ready-made solution for downloading the state of an entire object sub-tree at once. However, when only a fraction of the tree is useful to a client, the use of this interface causes unnecessary data transfer.

Participants

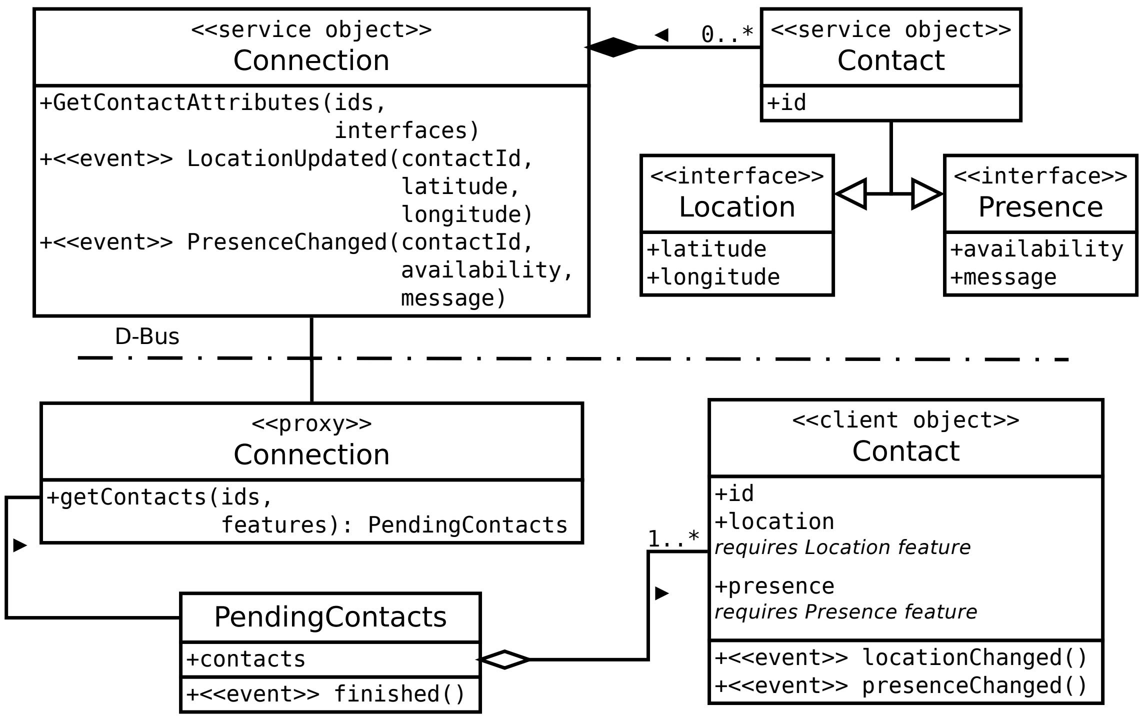
- Service data objects, of which multiple ones, but not all, are useful to an average client. The example in Figure 5.8 follows the use case from Telepathy, in which these object represent remote contacts.

- Service object acting as a multiplexer, which can give out data from a desired set of objects in a single operation. In the example, this is the Telepathy `Connection` service object.

- A proxy, which demultiplexes the batched data from the service objects. In the example, this is the proxy for the `Connection`.

- Client data objects, which the proxy uses to represent the demultiplexed service data. These objects represent individual contacts in the example.

Collaborations The proxy requests data from the multiplexing service object, when it is required. The data is represented to client code as separate data objects. The entire scheme is shown in Figure 5.8.



<<service object>>
Connection
+GetContactAttributes(ids, interfaces)
+<<event>> LocationUpdated(contactId, latitude, longitude)
+<<event>> PresenceChanged(contactId, availability, message)
0..*
<<service object>>
Contact
+id
<<interface>>
Location
+latitude
+longitude
<<interface>>
Presence
+availability
+message
D-Bus
<<proxy>>
Connection
+getContacts(ids, features): PendingContacts
PendingContacts
+contacts
+<<event>> finished()
1..*
<<client object>>
Contact
+id
+location
requires Location feature
+presence
requires Presence feature
+<<event>> locationChanged()
+<<event>> presenceChanged()

Figure 5.8: Structure of the Multiplexed State Synchronization pattern

Consequences The state of an arbitrary number of objects can be downloaded with a constant number of D-Bus method calls. Objects outside the interesting set do not increase the amount of downloaded data. However, as the proxy demultiplexes the data to independent objects, client code can still manipulate each object without regard to the batch in which they were actually downloaded in. As such, no complexity is added to the programming interface that is visible to client code.

6 AUTOMATED D-BUS PROXY GENERATION

At the lowest level, the details of the communication between proxies and services are tightly defined by the D-Bus interfaces, which the service objects implement. The client libraries have a corresponding low-level proxy layer, which is similar in principle to a collection of RPC stub procedures (see Section 3.1). In essence, a low-level proxy object provides a set of methods, which send the corresponding D-Bus `METHOD_CALL` messages to invoke remote methods, and events, which are fired, when D-Bus `SIGNAL` messages are received from the remote object. The low-level proxies always correspond one-to-one to D-Bus interfaces. The high-level proxies, which we explored in Chapter 5, utilize one or more low-level proxies to implement their communication with the remote object. This scheme is depicted in Figure 6.1.

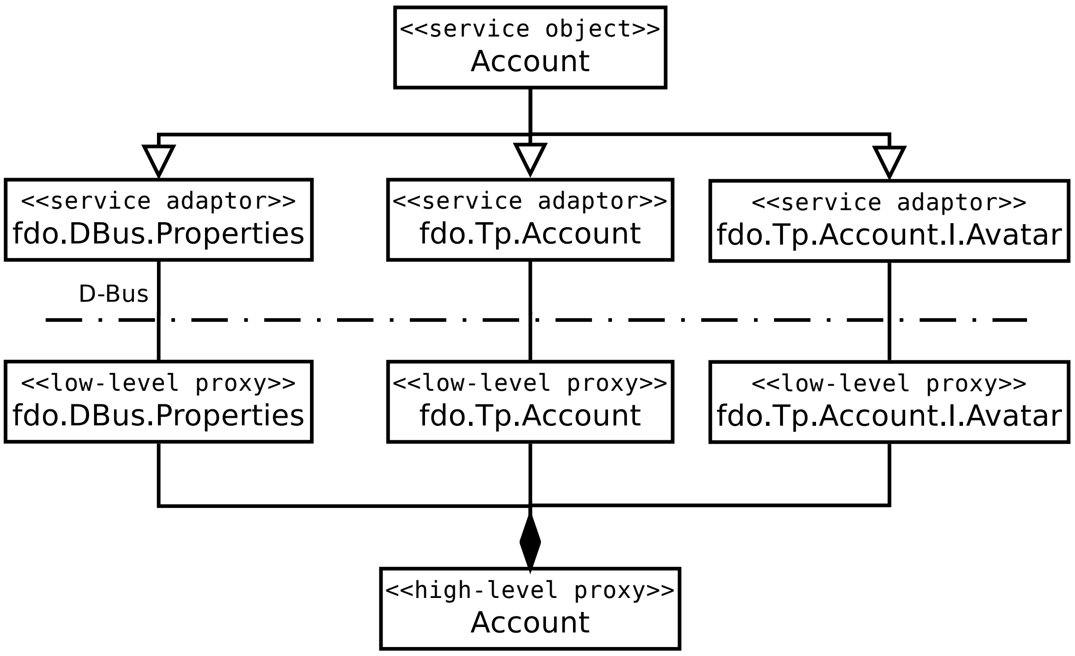


Figure 6.1: Layering of proxies in Telepathy client libraries

The close correspondence of low-level proxies and D-Bus interfaces allows generating the proxy code from sufficiently detailed interface descriptions. Thus, hand-writing these parts of the client libraries can be avoided. If the corresponding low-level service code is also generated from the same descriptions, it is guaranteed that the clients and the services will be compatible, as far as D-Bus messaging goes. In this chapter, we will describe the evolution of D-Bus interface description techniques, and what kind of client library code can be generated from descriptions written with them.

6.1 Procedural Interface Description

The `Introspectable` standard interface of D-Bus contains one method, `Introspect`, which returns an XML description of the object it has been called on. This description lists the methods, signals and properties in each interface that the object implements. The D-Bus type signatures of properties and the arguments of methods and signals are also included [PCL+12]. As an example, here is a description of the standard `Properties` interface (Section 3.4.4) in this format:

```
<interface name="org.freedesktop.DBus.Properties">
  <method name="Get">
    <arg name="Interface" type="s" direction="in" />
    <arg name="Property" type="s" direction="in" />
    <arg name="Value" type="v" direction="out" />
  </method>
  <method name="Set">
    <arg name="Interface" type="s" direction="in" />
    <arg name="Property" type="s" direction="in" />
    <arg name="New_Value" type="v" direction="in" />
  </method>
  <method name="GetAll">
    <arg name="Interface" type="s" direction="in" />
    <arg name="Properties" type="a{sv}" direction="out" />
  </method>
  <signal name="PropertiesChanged">
    <arg name="Interface" type="s" />
    <arg name="Changed" type="a{sv}" />
    <arg name="Invalidated" type="as" />
  </signal>
</interface>
```

The D-Bus specification suggests that this format could also be used as an *interface description language* (IDL), from which code can be generated [PCL+12]. The introspection information is sufficient to describe what kind of messages can be sent to an object, and what it sends out, as far as the D-Bus type system is concerned. Additionally, the names of interface members and arguments can be used as identifiers in proxy code, as long as some adjustment is performed to make them consistent with coding conventions in the target language. This can be sufficient for generating a useful proxy interface for very simple interfaces. For example, the `Set` D-Bus method could become the following C++-like proxy method declaration:

```
PendingOperation *set(const String &interface,
                      const String &property,
                      const DBusVariant &newValue);
```

For this simple method, this is a satisfactory interface. This is mostly due to its self-evident semantics, and the primitive types of the arguments. Let us consider a more complex example. The `ContactsChanged` signal of the `ContactList` interface is emitted on Telepathy `Connection` objects, when the contents of the user's server-stored address book changes. The signal is defined as

`ContactsChanged(a{u(uus)}: Changes,au: Removals)` [SIGNAL]

Representing the `ContactsChanged` signal in a C++ proxy is no longer straightforward. The first argument has a complex nested type—which C++ type should it be mapped to? And how should code which receives this signal interpret the integers and strings that make up that type? They do not even have names to explain themselves, after all. In the next section, we will describe a set of extensions to the D-Bus introspection format, which make it possible to solve these kinds of issues.

6.2 The Telepathy Interface Description Format

The D-Bus interfaces that Telepathy components implement are described with an extended version of the D-Bus introspection format [MMHT11]. These extensions make it possible to specify interface semantics in a more detailed fashion. The enhanced possibilities for code generation have led to others adopting [AFF+09] the use of this extended format as well.

6.2.1 Integer and string constants

The D-Bus type signature of the first argument of the `ContactsChanged` signal was `a{u(uus)}`. This means that it is a mapping from unsigned integers to structures, which consist of two unsigned integers and a string. The keys of the mapping are integer identifiers for contacts, and the value structure describes the disposition of the corresponding contact in the address book. The disposition tells if the remote contact has allowed us to see details such as whether they are online or away and so on, and if we have done the same for them. The structure of the entire data type is shown graphically in Figure 6.2.

As illustrated in the figure, the integers in the disposition structure are actually used to represent an enumerated type. Variables with an enumerated type can take one of just

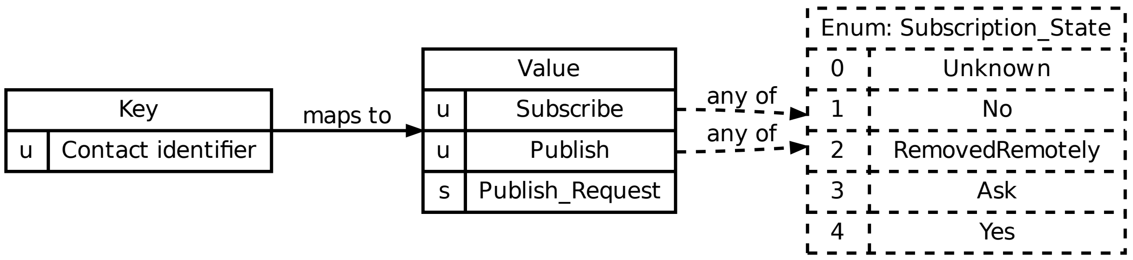


Figure 6.2: Layout of the first argument of the `ContactsChanged` signal

a few values, each of which has a distinct meaning. The Telepathy interface description format allows specifying the allowed values for enumerated types with the `tp:enum` element, as follows:

```
<tp:enum name="Subscription_State" type="u">
  <tp:enumvalue suffix="Unknown" value="0" />
  <tp:enumvalue suffix="No" value="1" />
  <tp:enumvalue suffix="Removed_Remotely" value="2" />
  <tp:enumvalue suffix="Ask" value="3" />
  <tp:enumvalue suffix="Yes" value="4" />
</tp:enum>
```

The code generator in the build system of the Telepathy-Qt library would turn this to the following C++ enumerated type declaration:

```
enum SubscriptionState {
  SubscriptionStateUnknown = 0,
  SubscriptionStateNo = 1,
  SubscriptionStateRemovedRemotely = 2,
  SubscriptionStateAsk = 3,
  SubscriptionStateYes = 4
};
```

A similar element, `tp:flags`, can be used to specify flag values for bit fields. In bit fields, a set of one-bit flag values are bitwise-OR'd together to produce a combination of flags represented as an integer.

D-Bus errors have names that resemble D-Bus interface names, e.g. `org.freedesktop.DBus.Error.UnknownMethod`. When a method produces an error, the error name is given as a string in the `ERROR` message. The `tp:possible-errors` element can be attached to a D-Bus method declaration to list the errors that can result from invoking that method, and to describe what meaning they have in that context. When standard D-Bus errors are not sufficient, the global `tp:errors` element can be used to define additional well-known error names, like:

```
<tp:errors namespace="org.freedesktop.Telepathy.Error">
  <tp:error name="Network Error" />
  <tp:error name="Not Implemented" />
</tp:errors>
```

The information in `tp:errors` elements can be used to define string constants for error names. The client libraries also include generated string constants for D-Bus interface names, as these have been found useful.

6.2.2 Complex types

The `tp:struct` element can be used to give a name and semantics to D-Bus structures. The definition of the contact disposition structure for the `ContactsChanged` signal follows:

```
<tp:struct name="Contact_Subscriptions">
  <tp:member name="Subscribe" type="u" tp:type="Subscription_State" />
  <tp:member name="Publish" type="u" tp:type="Subscription_State" />
  <tp:member name="Publish_Request" type="s" />
</tp:struct>
```

In this definition, the `tp:type` attribute is used to link the first two structure members to the enumerated type that was presented earlier. This attribute can also be used to reference a defined structure by its name. With this, we can define the full mapping type of the `ContactsChanged` signal's first argument, using the `tp:mapping` element:

```
<tp:mapping name="Contact_Subscription_Map">
  <tp:member name="Contact_Handle" type="u" />
  <tp:member name="States" type="(uus)" tp:type="Contact_Subscriptions" />
</tp:mapping>
```

With these type definitions, a C++ code generator could produce something like the following code to declare the corresponding C++ types:

```
struct ContactSubscriptions {
  SubscriptionState subscribe;
  SubscriptionState publish;
  String publishRequest;
};

typedef Map<unsigned int, ContactSubscriptions> ContactSubscriptionMap;
```

The `tp:type` attribute is also used to specify the semantical type to use for method and signal arguments and properties. The characters "`[]`" can be appended to any type name in a `tp:type` value to specify an array of the type to be used. For example, the Telepathy `ConnectionManager` interface defines a structure, which represents an allowed connection parameter, as follows:

```
<tp:struct name="Param_Spec">
  <tp:member name="Name" type="s" />
  <tp:member name="Flags" type="u" tp:type="Conn_Mgr_Param_Flags" />
  <tp:member name="Signature" type="s" />
  <tp:member name="Default_Value" type="v" />
</tp:struct>
```

This definition is used to declare the D-Bus property

`Parameters`: `a(susv)` [READ-ONLY PROPERTY]

on the `Protocol` interface, with the following XML code:

```
<property name="Parameters"
          type="a(susv)"
          access="read"
          tp:type="Param_Spec[]"
          tp:immutable="yes" />
```

The `tp:immutable` attribute shown in the above declaration can be used to mark a property as immutable, as defined in Section 5.2.

6.2.3 Miscellaneous features

Naming types and constant values based on their semantics makes D-Bus interfaces a great deal more self-documenting than if just the D-Bus type signatures were used. However, more elaborate free-form documentation is typically desirable. The `tp:docstring` element can be used to attach arbitrary textual documentation to almost any element in a D-Bus interface description, from a top-level interface to a single enumeration value. The documentation may also include XHTML markup, which can make longer runs of documentation more readable. While the Telepathy interfaces are thoroughly documented in this fashion, the `tp:docstring` elements were deliberately left out from the preceding interface description fragments to make them more concise.

The `tp:copyright` element can be used to specify the copyright ownership for a section of an interface description. Licensing terms can be written inside `tp:license` elements. The content of both of these elements is free text.

These additions make it possible to generate, besides library code, annotated documentation of D-Bus interfaces. The Telepathy D-Bus Interface Specification document [dTdCDG⁺12] is generated in this fashion from the interface description XML, with no other input data. The specification document is the canonical source of information on how components of the framework must communicate and behave.

7 CONCLUSION

In this thesis, we have analyzed the properties of the D-Bus system within the context of established research on object-oriented software design and inter-process communication. Practical considerations for the design and implementation of D-Bus-based object-oriented software have also been presented, based on these properties.

D-Bus was found to be a rather multi-faceted system. It supports both explicit invocation through method calls and implicit invocation via signals, and the match rules are a fairly complex example of a content-based publish/subscribe subscription system. This has made establishing a base of comparison for our analysis fairly challenging, as research on existing systems with each of these invocation patterns needed to be referenced. On the other hand, this multitude of features in D-Bus makes it possible to build elaborate mechanisms, such as our sophisticated state-caching proxy pattern (Chapter 5).

In Section 3.5, constraints for D-Bus messaging design were presented. The pseudo-blocking patterns for placing method calls were reasoned to be harmful for interactivity. Additionally, the classic pseudo-blocking pattern causes a loss of message ordering semantics and is prone to deadlocks, while the re-entrant variant makes class invariants unreliable. Fully asynchronous method calls do not have these drawbacks, but are programmatically unwieldy. Besides D-Bus, these conclusions would seem to apply to any IPC system with a call packet → reply packet style of procedure invocation (see Section 3.1).

Our presentation of the software design patterns in Chapter 5 has been mainly conceptual. Additionally, their possible effect on software complexity, performance and other non-functional qualities has been briefly reasoned about. These patterns used together would seem to reduce the complexity of inspecting and presenting the state of remote objects to the level of local objects, without violating the messaging constraints that were established earlier, or decreasing performance substantially. However, a case study of taking the patterns into use in an existing software system could provide empirical further proof for these claims. In any such effort, accuracy of performance and power consumption measurements will be affected by implementation details. For example, there are recent plans [Cre10] to reassign some message routing responsibilities from the D-Bus bus daemon to the operating system kernel. This alters the context switch requirements (Section 3.5.3) of D-Bus message transfer.

The design patterns presented in Chapter 5 and the interface description language described in Section 6.2 have materialized as a by-product of functional development of the Telepathy IM/VoIP framework. A study of adapting them in some other system could also provide insights on the breadth of their applicability. The multiplexed state synchronization mechanism from Section 5.5 likely needs the most adjusting work to fit into further uses besides communicating remote contact information in Telepathy.

As mentioned in Section 5.2, a basic property caching facility already exists in a generic form in the GDBus library. This facility could perhaps be augmented with support for custom change notification mechanisms and initialization of the state copy from known values, as present in the Telepathy client libraries. This would lower the barrier of employing these mechanisms in other software, but would likely require annotating their D-Bus interface descriptions with extra information, such as how properties can be marked as immutable in the Telepathy specification (Section 6.2.2).

We haven't often found the necessary delay between a service object state change and the corresponding proxy state cache update problematic. However, it requires some care to be taken in D-Bus interface design to avoid race conditions. Analyzing the consistency properties of the state caching pattern and its implications on interface design would be an interesting topic for future research.

In general, the Telepathy specification format (Section 6.2) has been found to be a sizable improvement over basic D-Bus introspection data for code and documentation generation purposes. It helps avoid hand-writing lower-level parts of D-Bus software and thus lowers implementation effort. However, this might be dependent on the target programming language; the suitability for generating code for other kinds of languages than the imperative C, C++ and Python remains to be seen.

A distributed software architecture built around an IPC system like D-Bus can have many benefits over a monolithic solution. However, as illustrated by this thesis, employing an IPC system optimally is far from trivial. Thus, the success of any such system is strongly dependent on its set of accompanying reusable implementation support libraries, code generation machinery and other tools which can aid and guide in this process.

References

[AAB+12] Sunil Mohan Adapa, Emanuele Aina, Ross Burton, Xavier Claessens, Alban Crequy, Guillaume Desmottes, Dafydd Harries, Philippe Kalaf, Alberto Mardegan, Robert McQueen, Simon McVittie, Mads Chr. Olesen, Pekka Pessi, Senko Rasic, Ole André Ravnaas, Olli Salli, Sjoerd Simons, Rob Taylor, Will Thompson, and Mikhail Zabaluev. telepathy-glib api reference. `http://telepathy.freedesktop.org/doc/telepathy-glib/`, 2012. Accessed April 26, 2012.

[AFF+09] Karl-Erik Ärzén, Pascal Faure, Gerhard Fohler, Marco Mattavelli, Alexander Neundorf, and Vanessa Romero. Interface specification. `http://www.actors-project.eu/wiki/images/a/a2/D1f-2009-01-31.pdf`, 2009. Accessed May 17, 2012.

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 26:92–109, April 1992.

[BA04] M.A.S. Boxall and S. Araban. Interface metrics for reusability analysis of components. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 40 – 51, 2004.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8:37–55, February 1990.

[BBG+63] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, et al. Revised report on the algorithmic language algol 60. *The Computer Journal*, 5(4):349–367, 1963.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.

[BCL+87] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *Software Engineering, IEEE Transactions on*, SE-13(8):880 – 894, aug. 1987.

[Ber00] Joseph Bergin. Multiple inheritance in java. `http://csis.pace.edu/~bergin`, June 2000. Accessed January 19, 2012.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2:39–59, February 1984.

[CCHO89] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 457–467, New York, NY, USA, 1989. ACM.

[CHC90] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.

[Coh84] A. Toni Cohen. Data abstraction, data encapsulation and object-oriented programming. *SIGPLAN Not.*, 19:31–35, January 1984.

[Cre08] Alban Crequy. Importance of the scheduling priority in d-bus. `http://www.maemonokian900.com/maemo-news/importance-of-the-scheduling-priority-in-d-bus/`, 2008. Accessed May 17, 2012.

[Cre10] Alban Crequy. D-bus in the kernel, faster! `http://alban-apinc.blogspot.fi/2011/12/d-bus-in-kernel-faster.html`, 9 2010. Accessed June 4, 2012.

[DMN67] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula 67 common base language*. Norwegian Computing Center, 1967.

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Some features of the simula 67 language. In *Proceedings of the second conference on Applications of simulations*, pages 29–31. Winter Simulation Conference, 1968.

[DSC92] A. Dave, M. Sefika, and R.H. Campbell. Proxies, application interfaces, and distributed systems. In *Object Orientation in Operating Systems, 1992., Proceedings of the Second International Workshop on*, pages 212 –220, sep 1992.

[dTdCDG+12] Daniel d'Andrada T. de Carvalho, Guillaume Desmottes, Julien Gilli, Dafydd Harries, Tobias Hunger, Andre Magalhaes, Robert McQueen, Simon McVittie, Mads Chr. Olesen, Senko Rasic, Ole Andre Vadla Ravnaas, Olli Salli, Sjoerd Simons, Raphael Slinckx, Rob Taylor, Naveen Verma, and Mikhail Zabaluev. Telepathy d-bus interface specification, version 0.26.0. `http://telepathy.freedesktop.org/spec`, 2012. Accessed June 4, 2012.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 254–269, New York, NY, USA, 2001. ACM.

[GG04] Nasib S. Gill and P. S. Grover. Few important considerations for deriving interface complexity metric for component-based systems. *SIGSOFT Softw. Eng. Notes*, 29(2):4–4, March 2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GN91] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91 Formal Software Development Methods*, pages 31–44. Springer, 1991.

[Goo75] John B. Goodenough. Structured exception handling. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '75, pages 204–224, New York, NY, USA, 1975. ACM.

[GS93] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 447–455, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Har08] Dafydd Harries. Telepathy rationale. `http://telepathy.freedesktop.org/wiki/Rationale`, 2008. Project wiki page. Accessed May 18, 2012.

[HD93] Y.-M. Huang and Y.-S. Duan. The design and implementation of a distributed object-oriented knowledge-based system for hierarchical simulation modeling. In *Proceedings of the 4th Annual Conference on AI, Simulation and Planning in High Autonomy Systems*, pages 164–170, Tucson, AZ, USA, 1993.

[HOD+11] Dafydd Harries, Mads Chr. Olesen, Guillaume Desmottes, Alban Crequy, Danielle Madeley, and Will Thompson. Telepathy developer notes: Tubes. `http://telepathy.freedesktop.org/wiki/Tubes`, 2011. Project wiki page. Accessed November 13, 2011.

[Hor90] Chris Horn. Is object orientation a good thing for distributed systems? In Wolfgang Schröder-Preikschat and Wolfgang Zimmer, editors, *Progress in Distributed Operating Systems and Distributed Systems Management*, volume 433 of *Lecture Notes in Computer Science*, pages 60–74. Springer Berlin / Heidelberg, 1990.

[Hug02] Elliott Hughes. How many trivial getter methods does java have? *SIGPLAN Not.*, 37:19–24, August 2002.

[JTM08] Matthew Johnson, Alp Toker, and Thiago Macieira. D-bus binding errors. `http://www.freedesktop.org/wiki/Software/DBusBindingErrors`, 2008. Accessed February 8, 2012.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 214–223, New York, NY, USA, 1986. ACM.

[Lin10] Linux man-pages project. *UNIX(7) - unix, AF_UNIX, AF_LOCAL: Sockets for local interprocess communication*, 2010.

[Lov05] Robert Love. Get on the d-bus. *Linux Journal*, 130, 2005. Also available as `http://www.linuxjournal.com/article/7744`.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the*

ACM SIGPLAN 1988 conference on Programming Language design and Implementation, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[Mad11] Danielle Madeley. Telepathy. In *The Architecture of Open Source Applications*. Lulu.com, 2011. Also available as `http://www.aosabook.org/en/telepathy.html`.

[Mar96] R.C. Martin. The interface segregation principle: One of the many principles of ood. *The C++ Report*, 8:30–36, 1996.

[McV08] Simon McVittie. Why you shouldn't block on d-bus calls. Technical blog post. `smcv.pseudorandom.co.uk/2008/11/nonblocking/`, 11 2008.

[Mey92] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40 –51, oct 1992.

[MMHT11] Robert McQueen, Simon McVittie, Dafydd Harries, and Will Thompson. Telepathy developer notes: Dbusspec. `http://telepathy.freedesktop.org/wiki/DbusSpec`, 2011. Project wiki page. Accessed May 11, 2012.

[NGGS93] D. Notkin, D. Garlan, W. Griswold, and K. Sullivan. Adding implicit invocation to languages: Three approaches. *Object Technologies for Advanced Software*, pages 489–510, 1993.

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 58–68, New York, NY, USA, 1993. ACM.

[PCL$^+$12] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, and David Zeuthen. D-bus specification, version 0.19. `http://dbus.freedesktop.org/doc/dbus-specification.html`, 2012. Accessed May 10, 2012.

[Pen07] Havoc Pennington. Best d-bus practices. `http://blog.ometer.com/2007/05/17/best-d-bus-practices/`, 5 2007. Technical blog post. Accessed May 16, 2012.

[Pen08] Havoc Pennington. Synchronous io never ok. `http://blog.ometer.com/2008/09/07/synchronous-io-never-ok/`, 9 2008. Technical blog post. Accessed February 9, 2012.

[Pet07] R.P.A. Petrick. Planning for desktop services. In *Proceedings of the 2007 International Conference on Automated Planning and Scheduling*, Providence, Rhode Island, USA, sep 2007. ICAPS. Available at `http://abotea.rsise.anu.edu.au/satellite-events-icaps07/workshop7/paper9.pdf`.

[RSW98] J.-P. Redlich, M. Suzuki, and S. Weinstein. Distributed object technology for networking. *Communications Magazine, IEEE*, 36(10):100 –111, oct 1998.

[S+00] D. Sweet et al. DCOP—desktop communication protocol. In *KDE 2.0 Development*, chapter 13. Sams, 2000.

[SB94] R. Scheifler and J. Brown. Inter-client exchange (ice) protocol, 1994.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, MA, USA, États-Unis, 1986. IEEE.

[SMM12] Olli Salli, Andre Magalhaes, and Simon McVittie. Telepathy-qt. `http://telepathy.freedesktop.org/doc/telepathy-qt/`, 2012. Reference documentation. Accessed April 24, 2012.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 38–45, New York, NY, USA, 1986. ACM.

[Tho11] Will Thompson. The slothful ways of d-bus. `http://willthompson.co.uk/talks/the-slothful-ways-of-d-bus.pdf`, 2011. Desktop Summit 2011 talk. Accessed April 27, 2012.

[Web01] Adam Brooks Webber. What is a class invariant? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 86–89, New York, NY, USA, 2001. ACM.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 168–182, New York, NY, USA, 1987. ACM.

[WFN90] E.F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 253 –259, may-1 jun 1990.

[Wit05] G. Wittenburg. Desktop ipc. `http://page.mi.fu-berlin.de/gwitten/studies/desktop_ipc.pdf`, 2005. Course essay. Accessed November 13, 2011.

[Wol92] M. Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *Software Engineering Journal*, 7(2):95 –101, mar 1992.

[XZ02] Liang Xianzhong and Wang Zhenyu. Event-based implicit invocation decentralized in ada. *Ada Lett.*, XXII:11–16, March 2002.

[Zeu12] David Zeuthen. Gdbus highlevel d-bus support. `http://developer.gnome.org/gio/unstable/gdbus-convenience.html`, 2012. Reference documentation. Accessed April 24, 2012.