servicerobotik

servicerobotik

autonome mobile Serviceroboter

Zentrum für angewandte Forschung
an Fachhochschulen

ACE/SmartSoft
Technical Details and Internals

Christian Schlegel
Alex Lotz

Hochschule Ulm
University of
Applied Sciences

Report 2010 / 01

Christian Schlegel
Hochschule Ulm
Prittwitzstrasse 10
89075 Ulm, Deutschland

schlegel@hs-ulm.de
http://www.hs-ulm.de/schlegel

Alex Lotz
Hochschule Ulm
Prittwitzstrasse 10
89075 Ulm, Deutschland

lotz@hs-ulm.de

29. November 2010

ACE/SmartSoft
Technical Details and Internals

Christian Schlegel
Alex Lotz

Contents

Chapter 1

Naming Service

1.1 Introduction and Overview

Communication patterns provide a link between a service requestor and a service provider. Although they can even be used to implement interfaces inside a component, both parts are normally located not only in different components, but the components are also distributed over different hosts. One therefore needs a scheme to locate and to address a specific service within a specific component.

The overall approach is shown in figure 1.1 and is described in detail in [2, section 5.4.5]. Each component has a unique name. Services provided by a component are denoted by names that are unique within a component. Service requestors do not have public names since they need not to be found from outside (they connect to service providers and are not contacted). Service requestors can become named ports by registering themselves at the wiring service (see [2, section 5.5.2.6]) of their component. As named ports, service requestors can be wired with service providers from outside the component. Port names also have to be unique within a component.

Services can now clearly be identified by a tuple {*component, service*}. A service requestor only needs to know this tuple to connect to a service provider. The wiring pattern is now based on fourtuples {*A:component, B:port, C:component, D:service*} to connect the service requestor port *B* of component *A* with the service provider *D* of component *C*. Of course, the framework establishes connections only to services compatible with the service requestor.

It now depends on the mapping of the SmartSoft communication patterns onto the underlying middleware how the above names of service providers used to establish connections between service requestors and service providers are being resolved. For example, the *CORBA* based implementation of SmartSoft relies on the *CORBA* naming service. In contrast, the ACE based implementation needs to provide its own naming service that relates the symbolic names for services to the addressing mechanism used inside ACE/SmartSoft.

The naming service of ACE/SmartSoft consists of two different parts. The first part

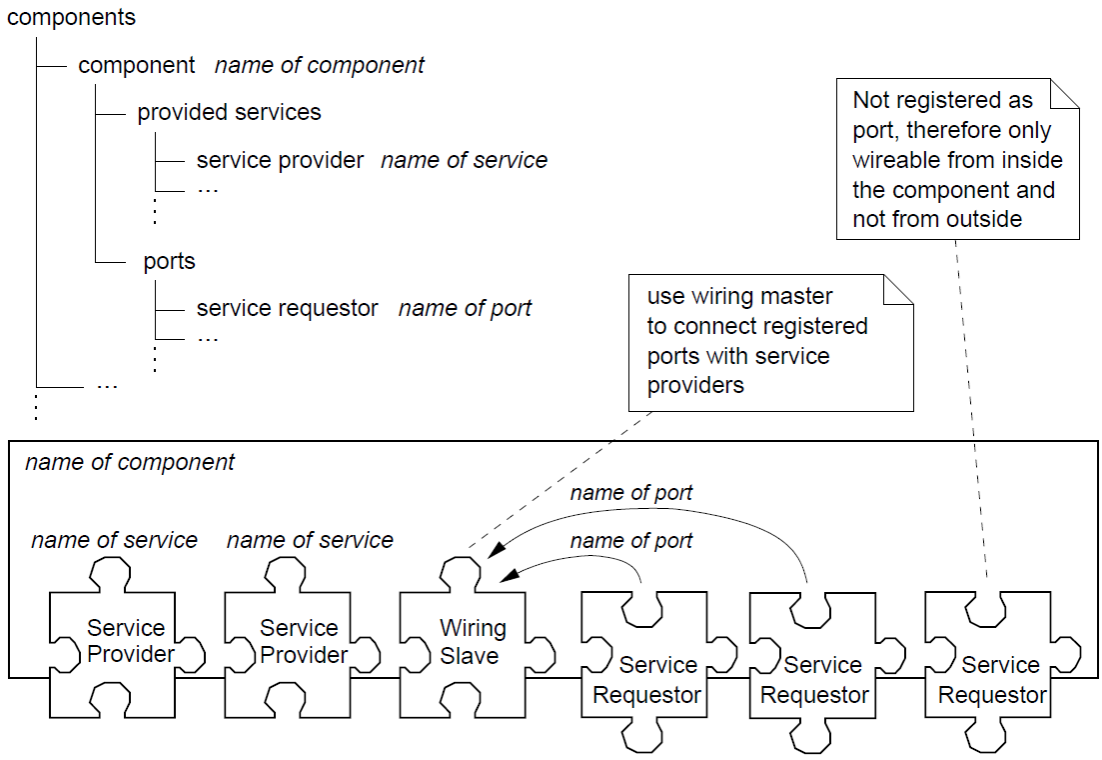

Figure 1.1: Naming of component and service

represents the naming service itself and is implemented as daemon. The daemon has to run in order to allow service requestors of components to resolve symbolic names of service providers they want to connect to. The second part is located inside components and provides the communication to the naming service daemon as required by the components. It provides a client access to the naming service daemon for a component.

After describing the basic structure of the ACE/SmartSoft naming service and how it is implemented on top of the classes provided by ACE, we first describe the details of the naming service daemon and then the details of the component internal part that handles the communication with the naming service daemon.

1.1.1 The ACE Naming Service

The ACE toolkit provides a set of core classes to implement a naming service [1, Chapter 21]. It comprises a persistent key/value mapping mechanism, which can, for instance, create a server-to-address mapping much like DNS but tailored to application's needs. The name space can cover a single process, all processes on a single node, or many processes on a network.

The main classes are[1]

[1] for reference, see README in `ACE_ROOT/netsvcs/lib`

- `Naming_Context`
  This class is the main workhorse of the Naming Service. A naming context instance is typically used to
  - bind or rebind a key to a value in the naming context,
  - unbind, or remove, an entry from the context,
  - resolve, or find, an entry based on a key,
  - fetch a list of names, values, or types from the context,
  - fetch a list of name/value/type bindings from the context.

  The naming context has two different modes of handling key/value mappings. The first one is to forward all actions to a remote name space (`Remote_Name_Space`). An internal ACE protocol behind the `Name_Proxy` interacts with the remote name space. The remote name space accepts that protocol via its `Name_Acceptor`. The second one is to process all actions on the local name space. The related key/value mappings are stored in a local file.

- `Name_Acceptor`
  The `Name_Acceptor` accepts connections from remote naming contexts and creates an instance of `Name_Handler` for each connection.

- `Name_Handler`
  All requests from the remote naming contexts are handled in the `handle_input` method of the `Name_Handler`. The requests are interpreted and are forwarded to the local namespace.

1.1.2 The ACE-Patch

The Problem

Releases of ACE up to the current stable release ACE 5.8.0 have an interoperability problem across platforms:

- a naming context accesses a remote name space via an ACE internal protocol. The problem is that the used data structure in the `Name_Request` is not marshalled using the regular ACE CDR Stream (which in fact are compatible across different platforms).

- Rather, for efficiency reasons, a lightweight marshalling is implemented in the `encode` and `decode` methods of `Name_Request`. A name/value/type set is just memcopied by the `Name_Proxy`. The getter/setter methods of `Name_Request` just use their local size of basic data types (`int` times `ACE_WCHAR_T`) for pointer arithmetic to calculate the start address of the name, the value and the type string in the local stream buffer.

- Both, the `int` and the `ACE_WCHAR_T` datatypes are of different size on different platforms (for example, size of `ACE_WCHAR_T` is 4 bytes on Linux and 2 bytes on Windows).

The Solution

The interoperability problem is solved by introducing a new class `Interop_Name_Request`. This class replaces `Name_Request`, has the exactly same interface and internal structure. The difference is that `ACE_WCHAR_T` is replaced by `ACE_TCHAR`. The latter is independent of operating system, byte order and 32/64 bit architecture. As result, the lightweight marshalling based on memcopies can be kept but now is platform independent.

The `Name_Proxy` normally uses the `Name_Request` which now is replaced by `Interop_Name_Request`. Thus, the data structure used within the ACE internal protocol for accessing remote name spaces is replaced by one that does not anymore show the above flaws. That replacement is achieved without degradation in efficiency and with only a minor modification that has no further effects on the ACE library.

The Implementation of `Interop_Name_Request`

The class `Name_Request` is defined in `Name_Request_Reply.h` shown in listing 1.1. The internally used buffer is defined in line 35 and depends on the platform specific representation of `ACE_WCHAR_T`.

```
1  // = The 5 fields in the <Transfer> struct are transmitted to the server.
2  // The remaining 2 fields are not transferred -- they are used only on
3  // the server-side to simplify lookups.
4
5  struct Transfer
6  {
7    /// Length of entire request.
8    ACE_UINT32 length_;
9
10   /// Type of the request (i.e., <BIND>, <REBIND>, <RESOLVE>, and <UNBIND>).
11   ACE_UINT32 msg_type_;
12
13   /// Indicates if we should block forever.  If 0, then <secTimeout_>
14   /// and <usecTimeout_> indicates how long we should wait.
15   ACE_UINT32 block_forever_;
16
17   /// Max seconds willing to wait for name if not blocking forever.
18   ACE_UINT64 sec_timeout_;
19
20   /// Max micro seconds to wait for name if not blocking forever.
21   ACE_UINT32 usec_timeout_;
22
23   /// Len of name in bytes
```

```
24      ACE_UINT32 name_len_;
25
26      /// Len of value in bytes
27      ACE_UINT32 value_len_;
28
29      /// Len of type in bytes
30      ACE_UINT32 type_len_;
31
32      /// The data portion contains the <name_>
33      /// followed by the <value_>
34      /// followed by the <type_>.
35      ACE_WCHAR_T data_[MAX_NAME_LENGTH + MAXPATHLEN + MAXPATHLEN + 2];
36    };
```

Listing 1.1: Name_Request_Reply.h

The class `Interop_Name_Request` shown in listing 1.2 is defined in `Interop_Name_Request.h`. As shown in line 36, the internally used buffer is now based on `ACE_TCHAR`. The following modifications have been made with the premise of avoiding any changes at the interface of `Name_Request` to ensure full replaceability:

- lines 18/19 in listing 1.2:
  the `ACE_UINT64` data type suffers from byte order problems that are not fully covered by the byte swapping mechanism inside the `encode` / `decode` methods. Thus, it has been replaced by an array of `ACE_TCHAR`.
- the byte swapping mechanism in `encode` and `decode` is not any longer needed when using `ACE_TCHAR` and is thus removed.
- another copy-constructor allows to copy a `Name_Request` onto a `Interop_Name_Request` and thereby handles the conversion of the internal data structure. This allows to seamlessly forward `Name_Request` objects to the new format.
- the constructor for creating an `ACE_Name_Request` message has the same interface as before (taking the same arguments without modifications of their data types). However, it now does the conversion onto the new internal data structure.

```
1   // = The 5 fields in the <Transfer> struct are transmitted to the server.
2   // The remaining 2 fields are not transferred -- they are used only on
3   // the server-side to simplify lookups.
4
5   struct Transfer
6   {
7     /// Length of entire request.
8     ACE_UINT32 length_;
```

```
9
10      /// Type of the request (i.e., <BIND>, <REBIND>, <RESOLVE>, and <UNBIND>).
11      ACE_UINT32 msg_type_;
12
13      /// Indicates if we should block forever.  If 0, then <secTimeout_>
14      /// and <usecTimeout_> indicates how long we should wait.
15      ACE_UINT32 block_forever_;
16
17      /// Max seconds willing to wait for name if not blocking forever.
18  //    ACE_UINT64 sec_timeout_;
19    ACE_TCHAR sec_timeout_[8];
20
21      /// Max micro seconds to wait for name if not blocking forever.
22      ACE_UINT32 usec_timeout_;
23
24      /// Len of name in bytes
25      ACE_UINT32 name_len_;
26
27      /// Len of value in bytes
28      ACE_UINT32 value_len_;
29
30      /// Len of type in bytes
31      ACE_UINT32 type_len_;
32
33      /// The data portion contains the <name_>
34      /// followed by the <value_>
35      /// followed by the <type_>.
36      ACE_TCHAR data_[MAX_NAME_LENGTH + MAXPATHLEN + MAXPATHLEN + 2];
37    };
```

Listing 1.2: Interop_Name_Request.h

Modification of the `Name_Proxy`

The definition of the class `Name_Proxy` is unmodified, only the implementation of the following methods of `Name_Proxy` has been changed slightly. Instead of `Name_Request`, we now use `Interop_Name_Request`. Figure 1.2 shows further details of `send_request` and `recv_reply`.

- `int send_request (ACE_Name_Request &request);`
  - in the original implementation, `encode` of `Name_Request` is called.
  - in the modified implementation, `encode` of `Interop_Name_Request` is called. Before calling `encode`, the parameter `request` of type `Name_Request` is converted into type `Interop_Name_Request`. The constructor of `Interop_Name_Request` accepts `request` as parameter.

- `int recv_reply (ACE_Name_Request &reply);`
  - in the original implementation, a received stream is directly stored in the `reply` parameter.
  - in the modified implementation, the received stream is put into a local object of type `Interop_Name_Request`. The `Interop_Name_Request` provides the method `convert_to` to convert the data structures into the expected type `Name_Request` of the `reply` argument.
- `int request_reply (ACE_Name_Request &request);`
  - this method is a combination of the above both methods. Since this method does not just call the above methods and instead reimplement this functionality, the above modifications are repeated here.
  - the ACE/SmartSoft naming service described in the next section requires access to the true TCP address of the socket underlying the `Name_Proxy`. The details are explained in the next section. A special mode of operation in this method is therefore introduced:
    - The `Name_Request` consists of *name, value* and *type*. Setting the *type* string to the value `smartip`, the `request_reply` method adds the IP address as colon-separated prefix to the *value*.

1.1.3 The ACE/SmartSoft Naming Service

The different libraries implementing the ACE/SmartSoft Naming Service are shown in figure 1.3. The ACE core library of the ACE software package shown on the left provides the core functionality to use the ACE naming service for customized applications. All our extensions to the core library are summarized as ACE patch. This patch is independent from SmartSoft and improves the ACE package with respect to cross-platform usage of its naming service core functionality.

The NS-lib library shown on the lower right comprises all functionality required to implement the naming service daemon of ACE/SmartSoft. The naming service daemon in its current implementation is a centralized service for all components. It provides a directory of available services and resolves names of required services [2, section 5.4.5].

The SmartSoft library shown on the upper right comprises all functionality required by the components of ACE/SmartSoft. Service providers and service requestors of components interact with the naming service to either announce their provided services or to resolve required services. The `SmartNamingHelper` provides access to that functionality of the `Naming_Context` needed in SmartSoft. It thereby parametrizes the generic `Naming_Context` for

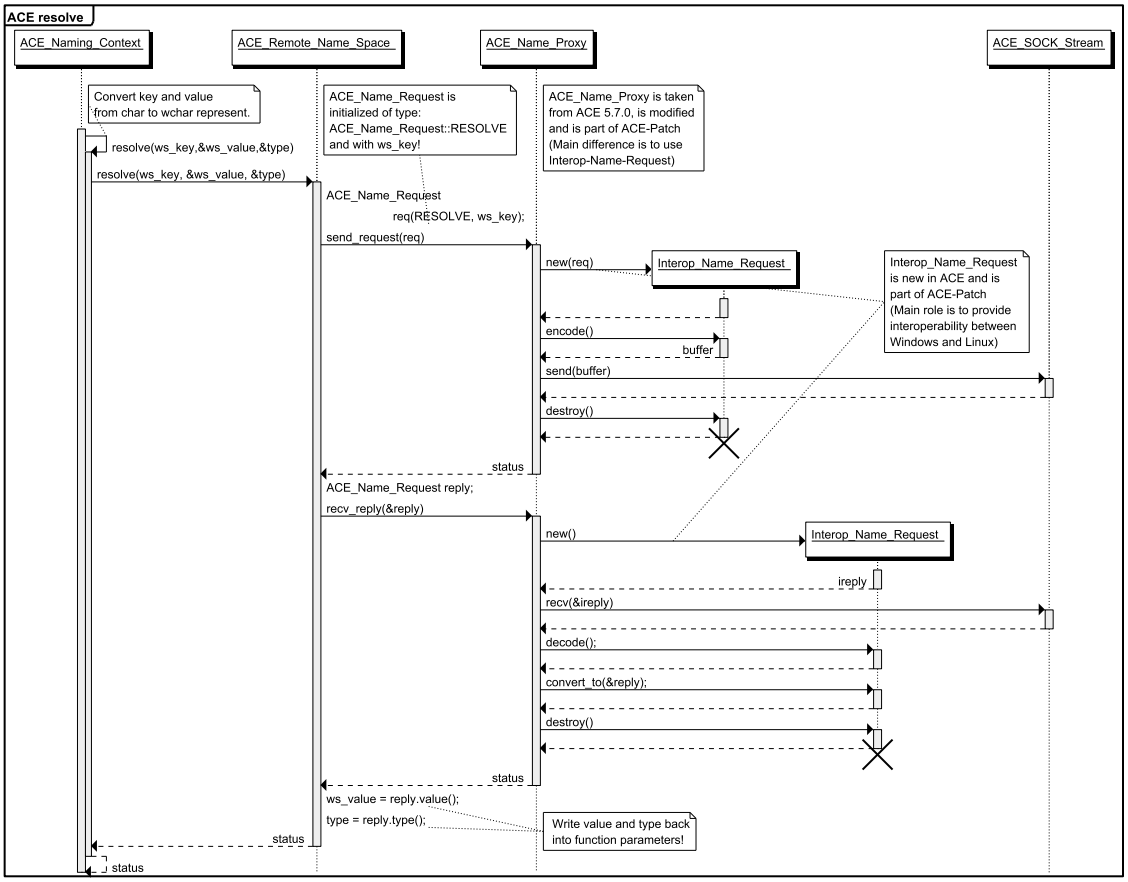

Figure 1.2: Interaction of `Name_Proxy` with `Interop_Name_Request`.

the requirements of SmartSoft. For example, the `Naming_Context` is configured to use a `Remote_Name_Space` instead of a local one.

The addressing scheme of SmartSoft is explained in detail in [2, section 5.6.6.4]. Service providers register themselves at the name service via a registration mechanism (*register*). Service requestors ask the name service to resolve service provider names (*query*). Since ACE/SmartSoft uses the message middleware patterns of ACE, the addressing scheme illustrated in [2, figure 5.87] is the one to be implemented here.

Name/Value entries of the Naming Service

The `Naming_Context` stores *name/value/type* triples. In ACE/SmartSoft, the name part in [2, figure 5.87] is stored in the name key, the address part in the value entry. Details are summarized in table 1.1.

SmartSoft requires a globally unique service identifier [2, page 166]. This can be composed out of the TCP socket address and a host-wide unique identifier. ACE/SmartSoft uses the ACE-class `UUID_Generator` to generate a host-wide unique identifier. This class im-

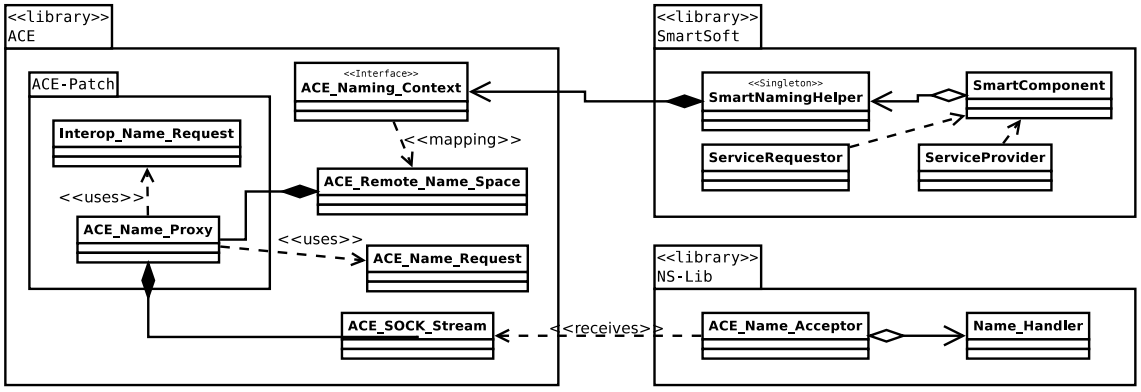

Figure 1.3: Overall structure of the SmartSoft Naming Service based on ACE.

name	<component name> <pattern type> (i.e PushNewest, Send, ...) <service name> <communication object name><...> (up to three are possible) all entries are of type ACE_TString (compatible to std::string)
value	<TCP socket address> (IP address + port number) <service identifier> (UUID) all entries are of type ACE_TString (compatible to std::string)

Table 1.1: The format of the name/value entry is exactly the same as specified in [2].

plements the specification of the INTERNET-DRAFT for UUIDs (Universally Unique Identifier) and GUIDs (Globally Unique Identifier).

Registration at the Naming Service

The registration is performed via the `smart_rebind` method of the `SmartNamingHelper`. It calls the `rebind` method of the `Naming_Context`. The `rebind` method overwrites values in case of existing names and otherwise creates a new name/value entry. We use a `rebind` instead of a `bind` due to the following scenario:

- in case a component crashes, its service providers might not correctly unbind or remove an entry from the naming context. As result, a no longer used name/value entry is still stored in the naming service.
- restarting this component results in registrations of its service providers. Since their name keys are still available in the naming service, one just has to overwrite their value entries. In case of using a `bind`, one would have first to clean up the naming service from outdated entries. Thus, using a `rebind` is a matter of comfort at the level of the internal implementation.

Outdated entries in the naming service can result in three different scenarios:

- Scenario 1:
  The name/value entries of service providers of a component are still listed in the naming service while the component already crashed and thus the services are not available anymore.

  Service requestors get the value entry and thus try to connect the TCP socket address of the service provider. The SmartSoft communication patterns detect that there is no end point available and handle that accordingly. Thus, dangling entries do not cause any problems in this scenario.

- Scenario 2:
  Same as scenario 1, but another and different service provider meanwhile caught the above TCP socket address. In this case, the naming service answers the request of a service requestor with the outdated value entry that now points to a completely different (and in many cases even incompatible) service provider. However, each *connect* of a service requestor transmits the globally unique service identifier – received from the naming service – to the service provider. The service provider detects that its service identifier and the one of the *connect* do not match. Thus, the connection request is rejected. Again, an outdated entry in the naming service cannot result in wrong connects.

- Scenario 3:
  Same as scenario 2, but the very same component has been restarted after a shutdown (or even after a breakdown without cleaning up the entries in the naming service). Since a restarted service provider might be in a different state than the previous instance of the service provider, SmartSoft expects that the replacement of the service provider is detectable.

  In case of ACE/SmartSoft, a loss of connection is detected by the used underlying ACE socket mechanisms. Detection is ensured by the used parametrization of the ACE mechanisms. Therefore, a disappearing service provider always enforces service requestors to reconnect which avoids wrong assumptions at the client side on states of service providers.

  Even in case the service provider under consideration happened to get its previous TCP socket address, the new service provider instance definitely possesses a different globally unique service identifier (in this example composed out of the same TCP socket address but a different service identifier). The enforced reconnect transmits the globally unique service identifier as received from the naming service. It only succeeds if the transmitted globally unique service identifier matches the one of the reached service

provider. Thus, independently of transient inconsistencies in the naming service, neither a service provider can seamlessly be replaced by another instance nor can incompatible services connect to each other.

Querying the Naming Service

The querying is performed via the `resolve` method of the `SmartNamingHelper`. This method is simply a wrapper around the `resolve` method of the `Naming_Context`. In case the name key exists in the naming service, the `resolve` method of `Naming_Context` returns the value of the corresponding name/value entry. However, the value is of type `ACE_NS_WString` which is a wide character string. Because SmartSoft uses ASCII strings internally, the value is converted inside of the `resolve` method of `SmartNamingHelper` into a `ACE_TString` which is compatible to the std::string (resp. ASCII string representation).

If a name key does not exist in the naming service, the `resolve` method of the `SmartNamingHelper` returns a corresponding error value (as described in Doxygen for this class).

The `resolve` method is the first step in the connection procedure of a service requestor. As described in the previous subsection, the value of a name/value entry in the naming service consists of a TCP socket address and a service identifier. The second step in the connection procedure is to create a socket connection to the service provider using the received TCP socket address. The last step is to transfer the service identifier to the service provider, which in turn compares it with its local (current) service identifier and in case of a match accepts or otherwise rejects the connection. In case a service provider accepts the connection, a success message is replied back to the service requestor, which then changes to the connected state. Otherwise a connection breaks up with a corresponding status code and the service requestor is in the disconnected state. That procedure is entirely the one described in [2, section 5.6.6.10].

As described in the previous subsection, a connection is only successful if a service provider and the service requestor are of the same type and the service identifier from the naming service is the same as in the service provider. All other cases – where for example the service identifier in the naming service is outdated – are handled correctly and lead to a consistent disconnected state.

1.1.4 Details of the Naming Service Classes

Figure 1.4 gives an overview of all classes which are used in the naming service on both sides, the naming service requestor and the naming service provider side. The class `SmartNamingHelper` (on the upper left) is implemented inside of the ACE/SmartSoft library and represents the naming service requestor.

The two classes, `Interop_Name_Request` and `ACE_Name_Proxy` are omplemented in the ACE patch, which is provided together with the ACE/SmartSoft library.

Finally, the two classes, `ACE_Name_Acceptor` and `ACE_Name_Handler` are implemented in the naming service daemon.

The interactions between all these classes is as follows. A call of one of the methods (like `rebind`) from the `SmartNamingHelper` (see table 1.2) internally rusults in a corresponding call of a method from the `ACE_Naming_Context` (as shown in the table). `ACE_Naming_Context` is initialized in this case to use the `ACE_Remote_Name_Space`, which result in a corresponding method calls in the `ACE_Name_Proxy` (see table).

The `ACE_Name_Proxy` in turn communicates with the `ACE_Name_Handler` class from the naming service daemon. The `ACE_Name_Handler` receives requests in its `handle_input` method from the remote `ACE_Name_Proxy` and dispatches the request to one of its internal methods (like `resolve`, `rebind`, etc.). These local methods are implemented to use a database (which is stored in a local file) to process the requests. The results from these requests are replied back to the `ACE_Name_Proxy`, then to the `ACE_Naming_Context` and finally to the `SmartNamingHelper` in ACE/SmartSoft.

Method in SmartNamingHelper	*Method in Naming_Context*	*Method in Name_Proxy*
smart_rebind	rebind	request_reply
unbind	unbind	request_reply
resolve	resolve	send_request + recv_reply

Table 1.2: Overview of the methods called in the `Naming_Context` configured with `Remote_Name_Space`

1.2 Naming Service Daemon

The naming service daemon provides a centralised directory service, which is used by components in ACE/SmartSoft to reqister their services and to query for service adresses respectivelly.

1.2.1 Overview on the Overall Structure

As introduced in section 1.1.4, the implementation of the naming service daemon is based on the two classes, the `ACE_Name_Acceptor` and `ACE_Name_Handler`. The overall structure of the naming service daemon uses the *Service Configurator Framework* of ACE (as desribed in [1, chapter 5]). As defined there, a dynamic service must be implemented as a shared library (`so` lib on Linux or `DLL` lib on Windows). This library is provided with the ACE/SmartSoft package inside of the *Utilities* folder in the *NS-Lib* project. This library is used in a separate programm to start the naming service as a daemon. This program is also provided with the ACE/SmartSoft package in the *Utilities* folder in the *NS-Daemon* project.

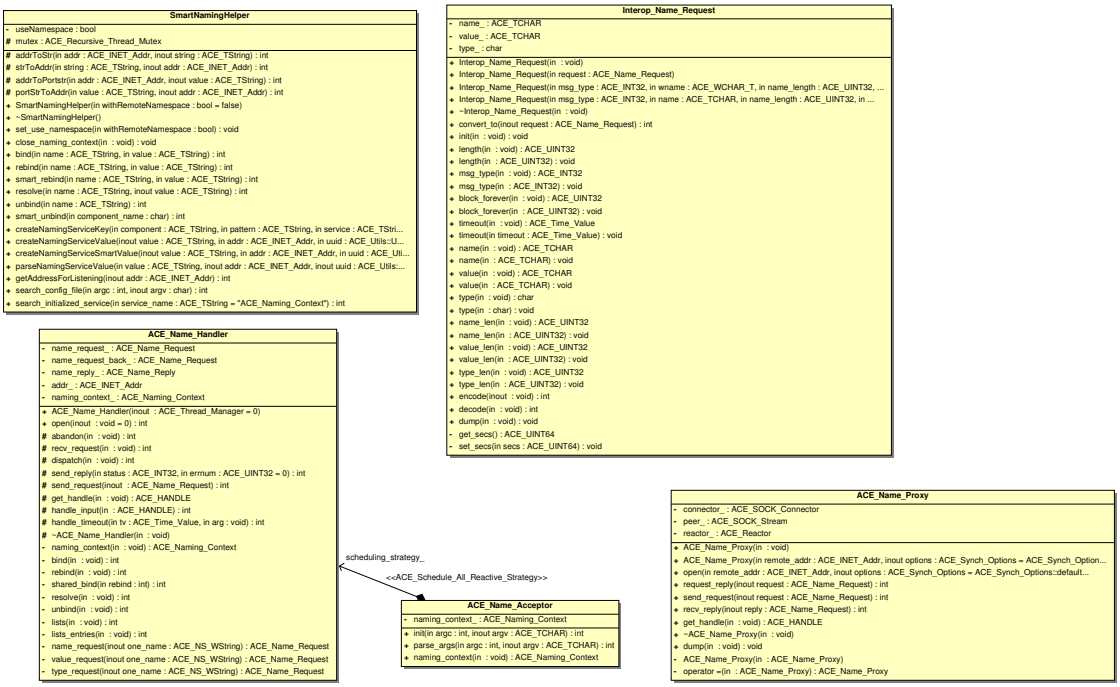

Figure 1.4: Class diagram of the SmartSoft Naming Service.

To start the naming service daemon, a configuration file is needed which is described in section 1.2.3 below. At the start procedure of the naming service daemon, first, the configuratino file is parsed. Second, the naming service library is loaded and is started as a service with the parameters from the configuration file.

Inside of the service the `ACE_Name_Acceptor` is initialised such, that new connectinos from naming service requestors can be accepted. For each accepted connection, the a new instance of a `ACE_Name_Handler` is created by the `ACE_Name_Acceptor`. This new instance then processes requests from its internal communication channel.

1.2.2 Naming Service Data Structures

The naming service daemon internally stores *name/value/type* tripples ina local database, which is stored in a local file and is thus persistent. For each service provider in ACE/SMARTSOFT, a tripple is stored in the database containing the name and the address of this service (as defined in table 1.1).

1.2.3 The Configuration File

The configuration file (shown in listing 1.3) presents an example for a typical configuration of a naming service daemon. As shown there the configuration is build-on two parts. The first part (shown in line 29 of listing 1.3) represents the instruction for the *Service Configurator*

Framework in ACE, for how to start the shared library. This line is not intended to by modified by component developers (framework users). The second part (shown in line 29 of listing 1.3) defines the parameter string. This string is used to customize the naming service daemon. For example the port number of the TCP socket address from the internal `ACE_Name_Acceptor` can be parametriezed without recompiling the application. Additionally, the directory and the name of the database file can be configured. Therefore, this string is intended to be modified by componet developers.

```
1  # These are the services that can be linked into ACE.
2  # Note that you can append the "netsvcs" with
3  # a relative path if you didn't set your LD search path correctly --
4
5  # ACE will locate this for you automatically by reading your LD search
6  # path.  Moreover, ACE will automatically insert the correct suffix
7  # (e.g., ".dll", ".so", etc.).  In addition, you can replace the
8  # hard coded "-p 20xxx" with "-p $PORTxxx" if you set your environment
9  # variables correctly.
10
11 # Following configuration assignment can be read as follows:
12 #1) dynamic: means the service will be started and configured dynamically
13     (with this script)
14 #2) Name_Server: is the class name used in the implementation
15     (this should not be modified)
16 #3) ServiceObject*: the type of Name_Server, its a ServiceObject of course
17 #4) NS-Lib: this is the library (DLL or so) file used for service init.
18 #   modify this Name, if libname changes.
19 #5) _make_ACE_Name_Acceptor(): is the constructor used to initialize Service
20 #   please do not modify this, otherwise Service wont work properly
21 #6) Parameter string passed to Service: these are the main configuration
22 # options to be modified:
23 #6.1) First parameter is the program's name (it is ignored in the service)
24 #6.2) -p: here is the port-number for the service to listen on
25 #         please modify this parameter on your needs
26 #6.3) -l: Folder Path, where Name-Database file should be stored,
27 #         this parameter is to be modified on your needs
28 #6.4) -s: Name for Name-Database file (modify on your needs)
29 dynamic Name_Server Service_Object * NS-Lib:_make_ACE_Name_Acceptor()
30   "main -p 8080 -l ./ -s MYNAMES"
```

Listing 1.3: Configuration file for Naming Service

1.2.4 Features and Characteristics

The implementation of the naming service daemon provides some error handling strategies during the initialisation of its service. For example at startup of the daemon, it is checked, whether the configuration file can be found either in the default search location (which is the

local directory, where the daemon is started), or the directory which is passed as parameter $-f$ during the startup of the daemon. If this file could not been found, or is not accessible (for example due to missing read flag on Linux), the naming service daemon prints a corresponding message on the console and abborts the start up procedure. The same occur if the shared library of the naming service is not accessible. Again, an error message is printed on the console and the start up procedure is abborted.

If the naming service daemon was able to initialize its internal service with the parameters from the configuration file, it prints a corresponding success message on the console.

At runtime the naming service daemon can handle different error situations. For example a broken connection to one (or some) of the components (i.e. because the component crashes down) do not cause the naming service daemon to crash. Instead the corresponding `ACE_Name_Handler` of the broken connection is cleaned up, and the naming service daemon proceeds its service for other connected requestors.

Finally, the content of the database inside of the naming service daemon can be shown by using the netsvc example from the ACE library. This example can be found in ACE under:

```
$ACE_ROOT/netsvcs/clients/Naming/Client
```

This tool can be started by using the same configuration file as is used for all components in ACE/SmartSoft.

1.3 Naming Service inside a Component

1.3.1 Overview on the Overall Structure

As mentioned above the naming service is divided into two main parts, the provider of a naming service and the requestors of the naming service. The provider is implemented as a daemon and a requestor is used in each component. The main work horse of the requestor is the class `SmartNamingHelper`. This class provides the main access point to the naming service internally in each component in ACE/SmartSoft. Additionally, this class provides some useful methods, that help for example to construct a name key and a value for the name/-value entries in the naming service (see Doxygen for more information on these methods). The class `SmartComponent` is the owner of this `SmartNamingHelper`. This means that the `SmartComponent` is responsible to initialize, respectively to destroy the `SmartNamingHelper`. Additionally, `SmartComponent` offers access to the `SmartNamingHelper` (through its public interface) for all the service requestors and service providers which are registered with this component. Thus, service providers can register, resp. remove its name/value entries in the naming service and service requestors can resolve name/value entries for particular services in the naming service.

Since the version 1.7.2 of ACE/SmartSoft, `SmartNamingHelper` is implemented as a singleton. Thus, several components can be initialized inside of the same process (in different

threads) and can use the same instance of the `SmartNamingHelper`. This saves resources and offers a generic usage of the naming service in ACE/SmartSoft. In particular this allows to map several components onto a single process as required in some operating systems.

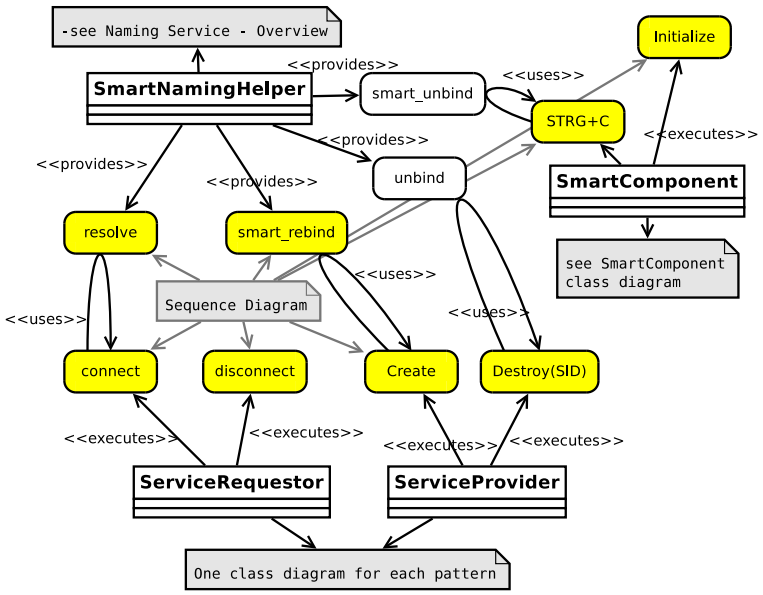

Figure 1.5: Overview on the SmartSoft Naming Service as part of each component (informal notion).

Figure 1.5 gives an overview of all interactions between a `SmartComponent`, its service providers/requestors and the `SmartNamingHelper`. This overview presents an informal and abstract view of the main parts which are of interest for the mapping of the SmartSoft idea onto ACE structures in the ACE/SmartSoft implementation.

In brief the interactions consist of the following parts. The main focus in `SmartComponent` is related to its initialisation and its destruction. For example, during initialisation, `SmartComponent` also initializes the `SmartNamingHelper`. During the destruction of the component, it cleans up its resources including the `SmartNamingHelper`. In case that during the destruction of a component one or some of its service providers crash down unexpectedly and are thus not able to remove their entries from the naming service, `SmartComponent` cleans up these entries as the last step before the component goes down. This is done by using the `smart_unbind` method of the `SmartNamingHelper`. As described in section 1.1.3, this is not really necessary, but leads to a cleaned up directory in the naming service to be more readable (if showing its contents).

The interactions between a service provider, resp. service requestor and the naming service are introduced in the next two subsections.

1.3.2 Communication Patterns: Service Requestor

As introduced in the previous subsection and illustrated in figure 1.5, the interactions of a service requestor with the naming service affects only the connection procedure. A connect internally uses the `resolve` method of the `SmartNamingHelper` to resolve the address to the service provider. This address is used in the connect method of the service requestor to establish the physical connection (internally based on a TCP socket in ACE/SmartSoft). The `resolve` sequence is illustrated in figure 1.6.

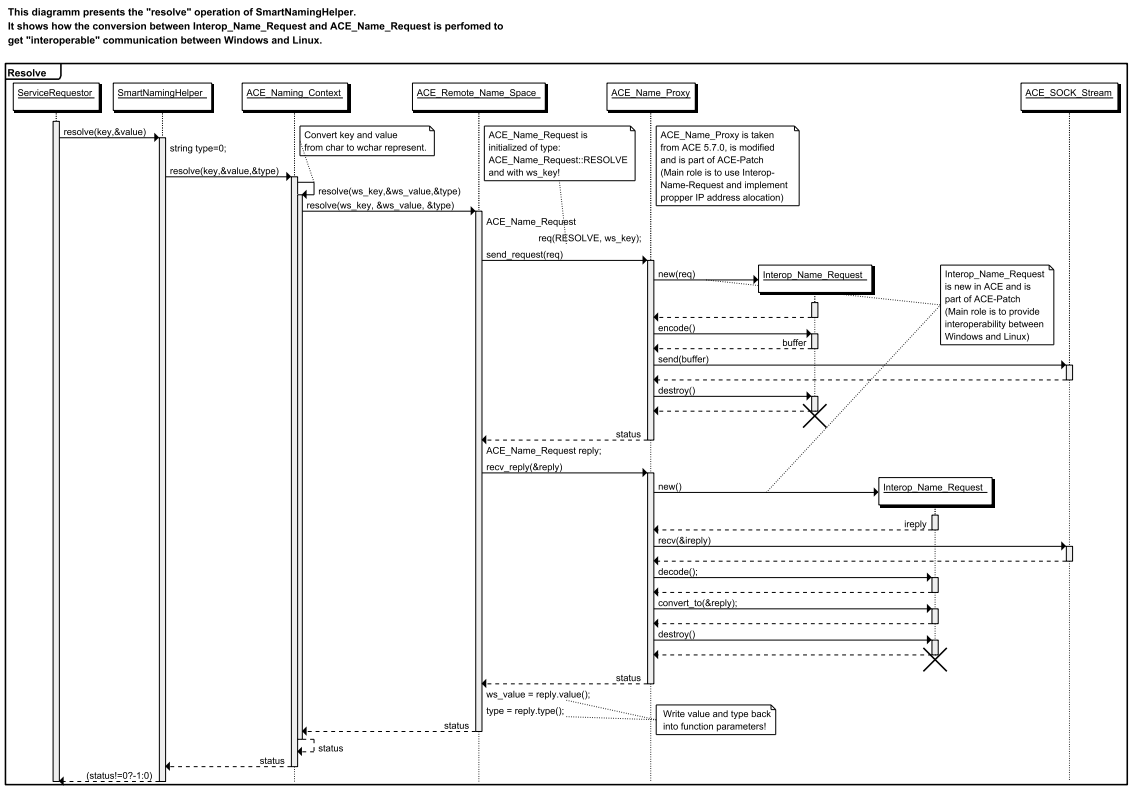

Figure 1.6: Sequence diagram: Resolve.

The sequence Diagram for the `resolve` method includes the usage of the ACE patch as described in section 1.1.2. As also shown in section 1.1.3 the `resolve` method comprises the `query` as defined in [2, figure 5.85]. As result, the `resolve` method returns either a null, which indicates successful resolution, or a value not equal to null, which indicates on a failed resolution. A failed resolution occur if the requested key is not available in the naming service or a communication error occurred. In case of a successful resolution the `value` parameter of the `resolve` method provides the string that contains the address to the service provider (as defined in table 1.1).

1.3.3 Communication Patterns: Service Provider

As illustrated in figure 1.5, the interactions of a service provider with the naming service affects only the creation and destruction procedures. A creation of a service provider results in a call of the `smart_rebind` method in the `SmartNamingHelper` class. The destruction of a service provider results in a call of the `unbind` method in the `SmartNamingHelper` class. As shown in table 1.2 both methods, `smart_rebind` and `unbind` results in the same call of the `request_reply` method of the `ACE_Name_Proxy` class. For this reason, it is sufficient to explain only one of the two methods, the other one works in the same way. Thereto the sequence of the `smart_rebind` method is illustrated in figure 1.7.

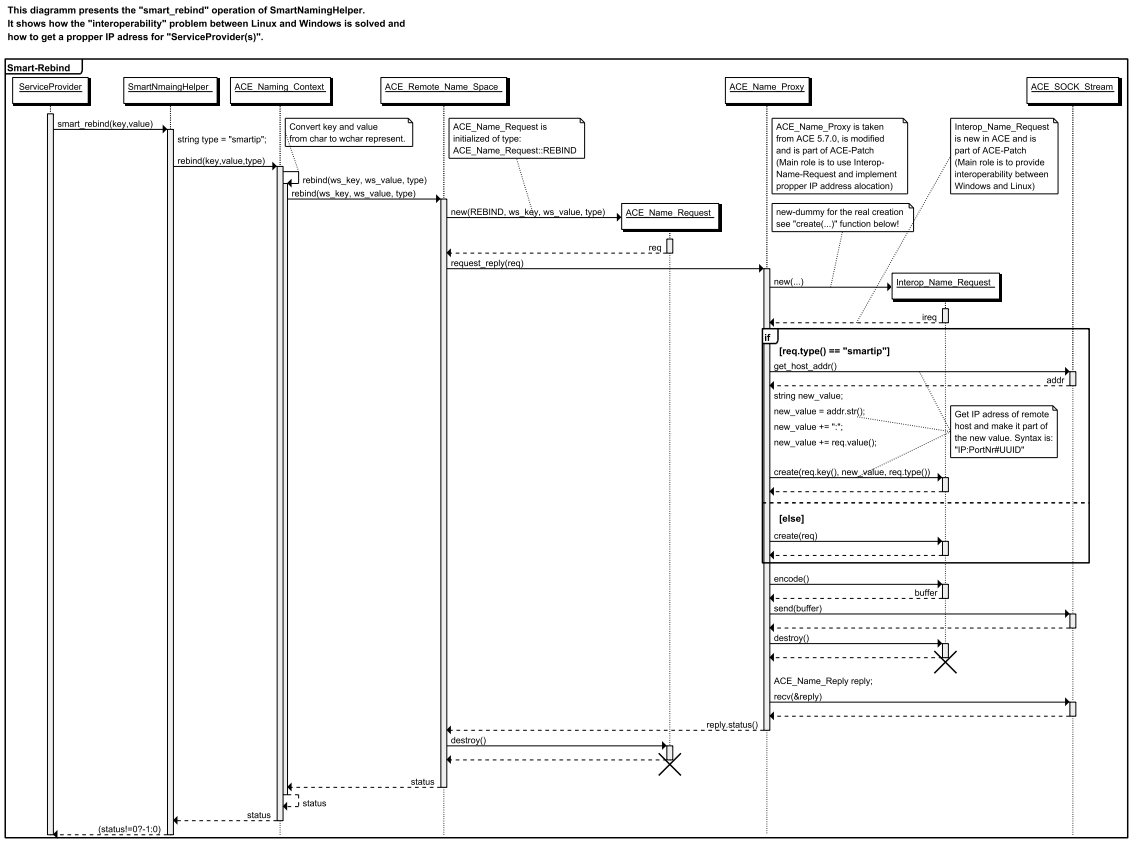

Figure 1.7: Sequence diagram: Rebind.

Again, the sequence Diagram for the `smart_rebind` method includes the usage of the ACE patch as described in section 1.1.2. As also shown in section 1.1.3 the `smart_rebind` method comprises the `register` call as defined in [2, figure 5.85]. Corresponding to the service requestor the service provider registers its name as the key and the address of its service as the value part. Additionally the type is set to the *smarip* string (as described in 1.1.2). This triple is used to register the service provider in the naming service. A service requestor can request for this entry in the naming service to establish connections with this service provider.

A `rebind` can fail only due to communication error. In this case it returns a value not equal to null. Otherwise a null is returned, which indicates successful registration.

1.3.4 The Configuration File of a Component

The configuration file (shown in listing 1.4) – that is required by each component – defines in principle how the class `SmartNamingHelper` should connect to the naming service daemon. Thereto the configuration file contains of two main parts. The first part is shown in line 24 of listing 1.4. Because the class `SmartNamingHelper` is based internally on the *service configurator framework* of ACE as described in [1, chapter 5], this line is necessary and is not meant to be modified by framework users. The subsequent line 25, however, represents the parameter string which can be modified by system developers (resp. framework users), depending on the system where this particular component is executed. If a component must be started with a different configuration, it is not necessary to recompile this component. A restart reads again the configuration file.

```
1  # Note that $PORT is an environment variable that is
2  # automatically interpreted and substituted by ACE!
3  # static ACE_Naming_Context "main -p $PORT -h tango"
4
5
6  # Following configuration assignment can be read as follows:
7  #1) dynamic: means the service will be started and configured dynamically
8      (with this script)
9  #2) ACE_Naming_Context: is the class name used in the implementation
10     (this should not be modified)
11 #3) ServiceObject*: the type of ACE_Naming_Context, its a ServiceObject
12 #4) ACE: this is the library (DLL or so) file used for ClientSide service
13 # modify this Name, if libname changes.
14 #5) _make_ACE_Name_Acceptor(): is the constructor used to initialize Service
15 # please do not modify this, otherwise Service wont work properly
16 #6) Parameter string passed to Service: these are the main configuration
17 # options to be modified:
18 #6.1) First parameter is the program's name (it is ignored in the service)
19 #6.2) -p: here is the port-number used to find NamingService
20 # please modify this parameter on your needs
21 #6.3) -h: host address where NamingService is running
22 # please modify this parameter on your needs
23 #6.4) -c: Lets service search for NamingServie net-wide (do not modify this)
24 dynamic ACE_Naming_Context Service_Object * ACE:_make_ACE_Naming_Context()
25   "main -p 8080 -h 127.0.0.1 -c NET_LOCAL"
```

Listing 1.4: Configuration file for a component

Chapter 2

Mapping of the Communication Patterns

All communication capabilities of each component in ACE/SmartSoft are mainly based on the `Reactor` pattern [3, chapter 3], which provides event handling and event demultiplexing mechanisms inside of components. The `Reactor` pattern is implemented in the `ACE_Reactor` class in the ACE library. That is, the `ACE_Reactor` implements an event based client-server functionality that minimizes required resources. For example, an `ACE_Reactor` needs exactly one thread of control, which is typically the main thread in a component. Each component internally uses exactly one `ACE_Reactor` instance (see Figure 2.1). This instance is responsible to react on incoming events from the underlying communication mechanism and to call appropriate handlers inside of service providers and service requestors which are attached to this component. Therefore, the Acceptors, Connectors and ServiceHandlers (see below) from communication patterns are registered in the Reactor. Thus, the Reactor can call the handler methods of these classes.

The communication patterns of SmartSoft use a message protocol for interaction that can be mapped onto all kinds of different middleware systems with all kinds of different characteristics. The *connection oriented split protocol* as described in [2, section 5.6.6] only requires that messages between a particular client and a particular server keep their initial order and never pass each other [2, page 137]. In particular, all internal interactions of the communication patterns are based on the (C^*/U) interaction pattern (see [2, section 5.6.6.2] that requires only one-way messages and works with a reliable send policy, a delivered policy and even a processed policy.

ACE provides different mechanisms to implement a message based system and due to the flexibility of the SmartSoft protocol, different mappings are possible. The ACE/SmartSoft implementation uses a straight-forward mapping to keep the overall complexity of the communication layer as low as possible. The reactor / acceptor / connector patterns of ACE allow for a message based communication with a delivered policy where messages keep their

order. Therefore, the communication layer of ACE/SmartSoft and its messages can be directly mapped onto this generic ACE communication mechanisms.

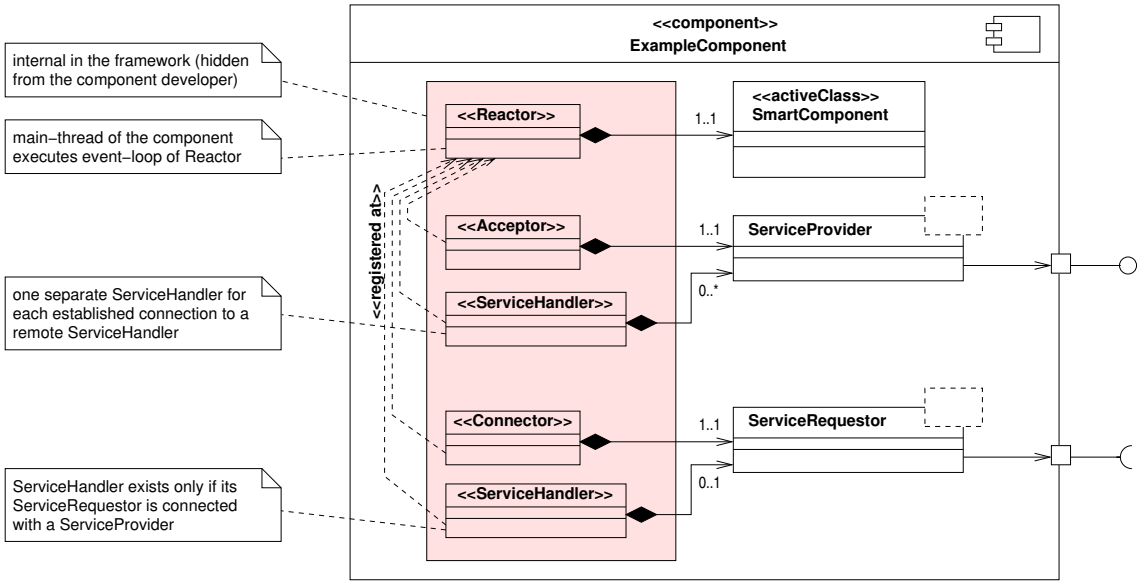

Figure 2.1: The internal core structure of a component in ACE/SmartSoft.

The *connector-acceptor* pattern [3, chapter 3], is part of the ACE library. Each service requestor uses an own instance of an `ACE_Connector` and each service provider uses an own instance of an `ACE_Acceptor` (see Figure 2.1).

The `ACE_Svc_Handler` class from the ACE library provides a connection oriented and platform independent peer-to-peer communication between two endpoints. It can be configured to be a server, a client, or even both (by providing bidirectional communication). ACE/SmartSoft provides the class `ServiceHandler`, which is derived from the `ACE_Svc_Handler` class. The responsibilities of a `ServiceHandler` are twofold. First, this class hides the complexity of `ACE_Svc_Handler` and parametrizes it to perform bidirectional communication. Thus, a `ServiceHandler` in ACE/SmartSoft is able to send messages to a remote endpoint and to receive messages from a remote endpoint via a handler method. Second, this class implements the glue logic between the events in the `ACE_Reactor` and the callback methods inside of the communication patterns in ACE/SmartSoft.

Figure 2.2 shows further details of the structures as described above. Each service requestor internally creates a `ServiceHandler` during its connection procedure. A service provider also internally creates a `ServiceHandler` for each connected service requestor. Thus, a point-to-point connection (internally based on TCP sockets) is created. Each `ServiceHandler` acts as a sender and as a receiver at the same time. Thus, on the one side (depending on the communication pattern) a `ServiceHandler` sends a message and on the other side a corresponding `ServiceHandler` receives a message, calls an appropriate callback method

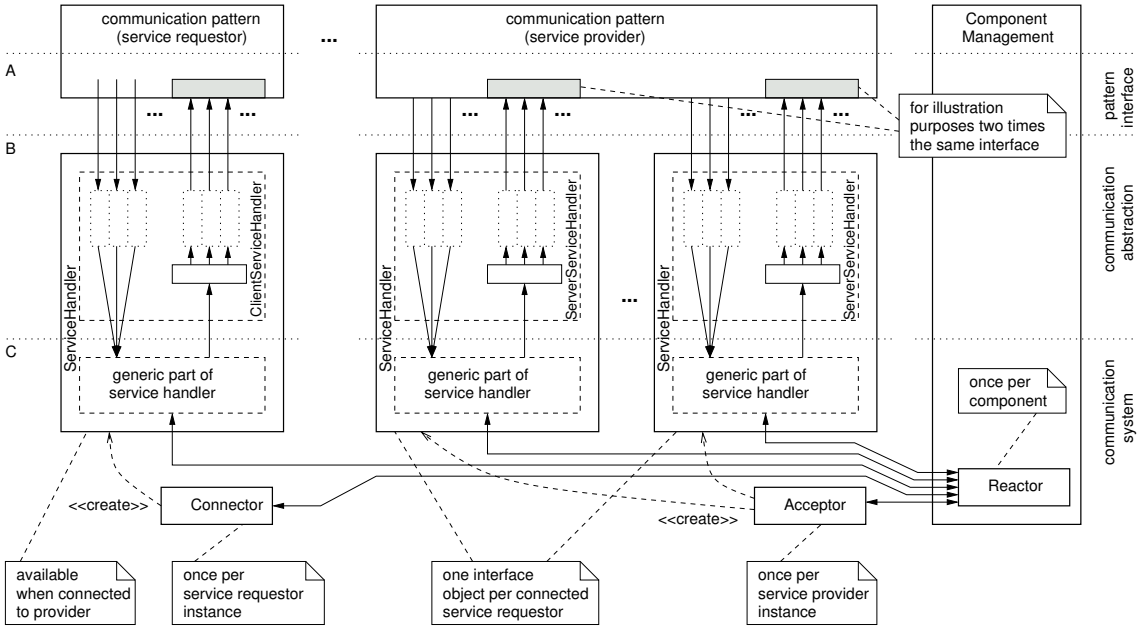

Figure 2.2: The Interface Objects implemented as ServiceHandler.

of the communication pattern and passes the message content to this callback method as a parameter. This complies to the *Interface Objects* for incoming messages as described in [2, page 161].

2.1 Service Requestors

An instantiation of a service requestor in ACE/SmartSoft causes an instantiation of an internal `Connector`. The `Connector` of the service requestor is then ready to be connected to an `Acceptor` which is available in a compatible service provider. A physical connection between a service requestor and a service provider is available first in the connection procedure and is destroyed in the disconnection procedure. Most of the details related to the mapping of service requestors to one particular communication middleware can be shown by describing the connection and disconnection procedures as shown in the following two subsections.

2.1.1 Connect

The whole connection procedure that is generic for all service requestors in ACE/SmartSoft is shown in the sequence diagram in figure 2.3.

First, a key is created (using the method `createNamingServiceKey` from `SmartNamingHelper` class) in the service requestor which represents the name of a service provider to which the service requestor wants to connect. After that, this key is resolved (queried) in the naming service. A successfully resolved key provides a TCP socket address to the remote

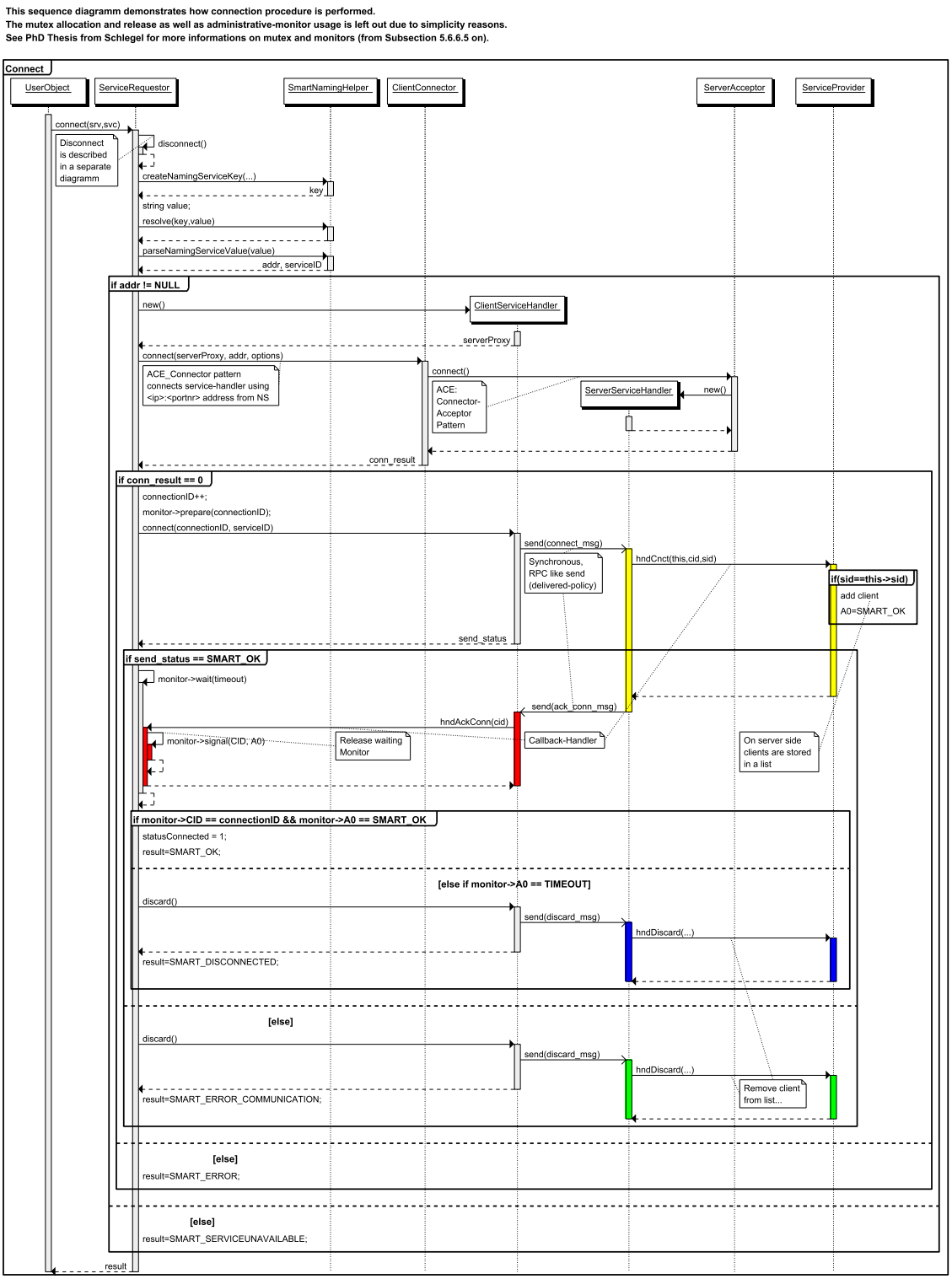

Figure 2.3: Sequence diagram: Connection in a Service Requestor.

`Acceptor` of the corresponding service provider. Additionally to the TCP socket address a globally unique service identifier is returned which is later transmitted to the service provider which checks if the service identifier is valid and accepts or rejects the connection.

Next, the physical connection between a service requestor and a service provider is established by using the connector-acceptor pattern in ACE. For this purpose an instance of a local `ServiceHandler` is created inside of the service requestor. This service handler is connected to a remote endpoint by using the `connect` method in the local `Connector`. A remote `Acceptor` creates a corresponding endpoint - which is again a local instance of a `ServiceHandler` - on demand.

Now, the connection channel is created and messages can be exchanged between the two endpoints. From now on, the regular connection routine, which is independent of the underlying communication mechanism, is continued. In brief, a service requestor generates a connection id and sends a message containing this id and the service id to the service provider. The service requestor then blocks on its internal `AdministrativeMonitor` (see [2, figure 5.92]) until the service provider replies on this message. A service provider validates the service id and accepts or rejects the current connection, by sending a corresponding message back to the service requestor. A service requestor now can switch into the connected state and return with a corresponding status code. This procedure is described in detail in [2, page 158].

2.1.2 Disconnect

A complete disconnection procedure is shown in the sequence diagram in figure 2.4.

The first part of the disconnection procedure is independent of the underlying communication middleware. In brief, a service requestor sends a disconnect message to the service provider, which immediately acknowledges the corresponding disconnection.

At this point a middleware specific problem occur. The destruction of the two `ServiceHandler`, one in the service provider and one in the service requestor, must be carefully implemented. The default behavior of the class `ACE_Svc_Handler` (resp. `ServiceHandler`) is as follows. If one of the two endpoints in a point-to-point connection is destroyed, the other endpoint automatically destroys itself also. This is reasonable on the first view, because this guarantees that the resources are always cleaned up correctly, even in the cases where the connection breaks down for different reasons. However, if an instance of this class is referenced using pointers it can happen that this class cleans up itself and the pointer points into void.

The solution for this problem is that the responsibility - to clean up the `ServiceHandler` class - must be transferred to the superordinate communication pattern. The reason is that only inside of the communication pattern it can be decided when a `ServiceHandler` is safe to be removed and when the corresponding pointers are not used any more. This behavior can be implemented by overloading the `handle_close` callback method from the base class `ACE_Svc_Handler` in the derived class `ServiceHandler`. The implementation of the `han-`

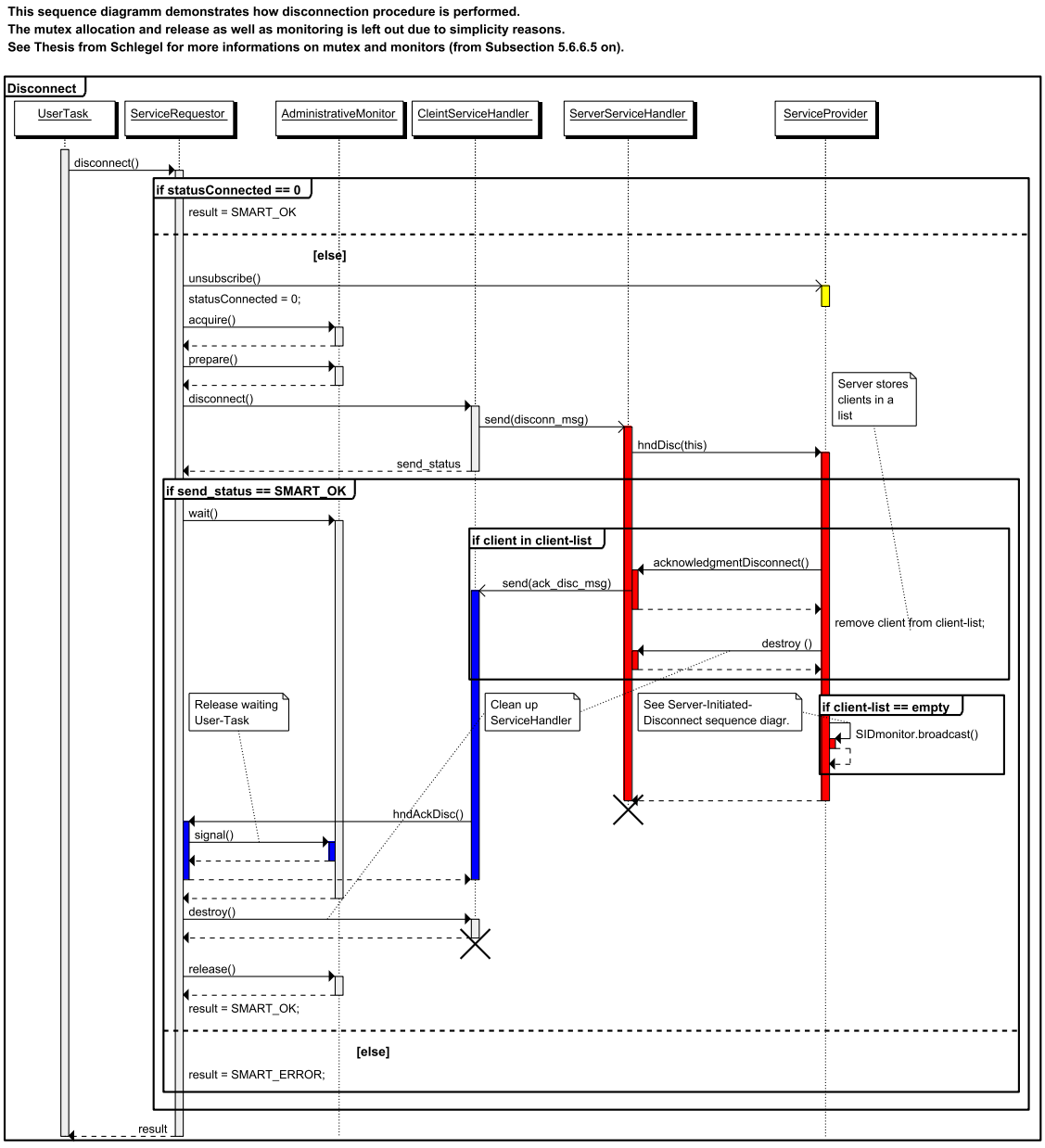

Figure 2.4: Sequence diagram: Disconnection in a Service Requestor.

`dle_close` method is changed such that only the internal socket connection is closed, but the class remains accessible. The closure of the socket connection is necessary in the case where the connection breaks down unexpectedly due to an error in the network. Even in this case it is safe to close the internal socket, as shown in the following two scenarios.

- If a communication pattern tries to send a message using its `ServiceHandler` whose internal socket connection meanwhile broke down, the send method returns immediately with a return value not equal to null. This value is evaluated by the communication pattern and a corresponding status code is returned.
- Even if the socket connection breaks down while a communication pattern is transmitting a message it is noticed in the communication pattern and never leads to an unexpected behavior. Thus, the delivered policy is always guaranteed.

With this customized behavior the regular destruction of the two `ServiceHandler` is performed as follows. After the service provider has acknowledged the disconnection, it is safe to remove the `ServiceHandler` on this side (because of the delivered policy). The service requestor on the other hand receives the acknowledge-disconnect message and must first process the `hndAckDisc` callback method before its `ServiceHandler` is also destroyed (see figure 2.4).

2.2 Service Providers

An instantiation of a service provider in ACE/SmartSoft causes an instantiation of an internal `Acceptor`. This `Acceptor` is initialized with the `Reactor` instance from the component, to which this service provider belongs to (resp. is attached to). Each `Acceptor` gets an own TCP socket address (which is managed by the underlying OS). This TCP socket address is used by `Connectors` in the network.

The lifetime of an `Acceptor` is directly linked to the lifetime of its superordinate service provider. Because the connection and disconnection procedures are already described in the foregoing section, in the following, the creation and destruction of a service provider is explained.

2.2.1 Creation of a Service Provider

The creation of a service provider is an important procedure that has some points which are individual according to the mapping of the service provider to one particular communication middleware. The whole initialisation procedure is shown in the sequence diagram in figure 2.5.

The `Acceptor` inside of the service provider gets an own TCP socket address. This address contains of a IP address and a port number. The port number is automatically chosen by

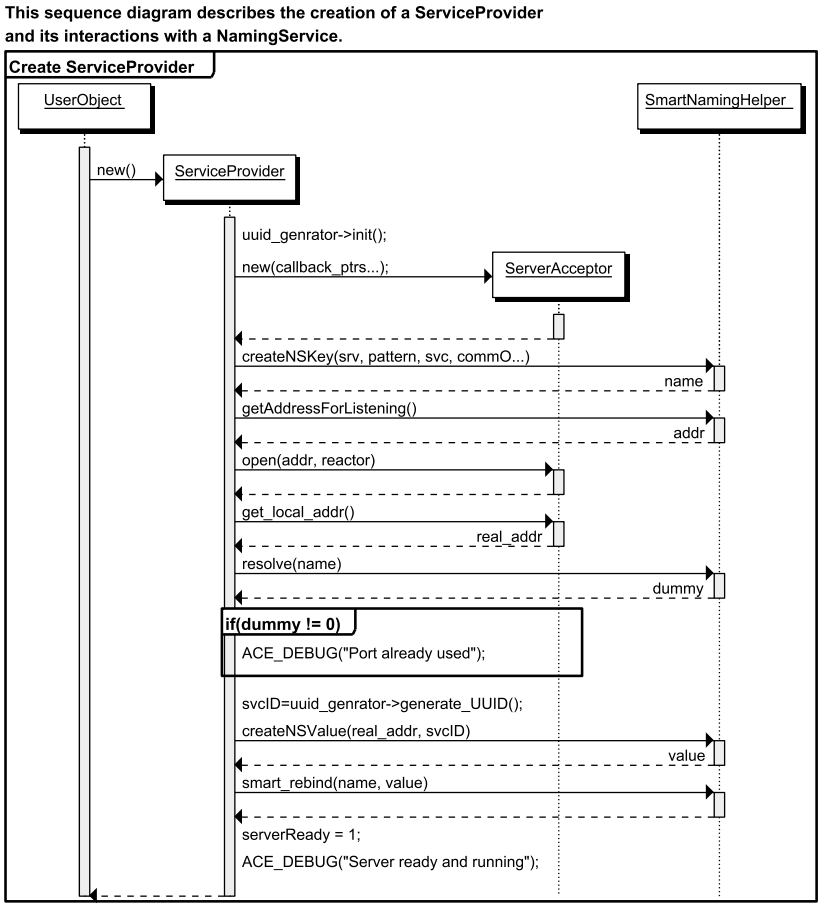

Figure 2.5: Sequence diagram: Creation of a Service Provider.

the underlying OS from one of the currently unused port numbers. Additionally to the TCP socket address the service provider generates a globally unique identifier, which represents a service identifier that is unique over time and space.

The creation of the service provider is as follows. First a key is created, which represents a unique name in the system for the service. This name is completely independent of the underlying communication middleware. After that the `Acceptor` is created using the next free port number. Next, a key/value entry is created that will be stored in the naming service. The entry consists of a key being a middleware independent service name, and a value that consists of the TCP socket address of the `Acceptor` and the service identifier. This entry is stored in the naming service by using the `smart_rebind` method of the `NamingHelper` class.

In the case where the key is already available in the naming service, a warning message is printed on the console and the value of the key is overwritten in the naming service. As described in 1.1.3 it is always safe to rewrite (resp. rebind) entries in the naming service.

2.2.2 Destruction of a Service Provider

The sequence diagram for the complete destruction procedure of a service provider is shown in figure 2.6. The destruction of a service provider consists of the following two main parts. First, the service provider asks all service requestors (which are currently connected to it) to disconnect. Therefore the server-initiated-disconnect subprocedure is used. Second, after all service requestors are successfully disconnected from this service provider, it removes its name/value entry from the naming service and cleans up its internal resources.

The first step during the destruction of a service provider is to reset its `serverReady` flag to null and to deactivate its `Acceptor` handler (see figure 2.6). This ensures, that from now on, no new connections can be established and the internal list of the interface objects remains stable. With that, it is possible to ask all currently connected service requestors to disconnect from this service provider. This is done in the second step by using the server-initiated-disconnect (short SID) procedure.

The SID procedure is independent of the mapping to the underlying communication middleware and is described in detail in [2, pages 193-195]. A mentionable detail is the realisation of the *active queue* inside of the component management. The active queue decouples the SID requests on the service requestors side. In ACE/SmartSoft the active queue is implemented inside of the `SmartComponent` in the `SIDHandler` class.

Finally, after all clients are disconnected successfully, the key/value entry for this service provider is unbound (removed) from the naming service and the rest of the resources are cleaned up.

2.3 Further Details on the Transportation of Data

In the foregoing two sections the mapping of communication patterns on the ACE communication middleware are described with the main classes `Reactor`, `Connector-Acceptor` and `ServiceHandler`. This mapping focuses on the interactions and coordinations between the communication patterns and the communication middleware. That is, service providers and service requestors now can interact with each other using the capabilities of ACE. However, the communication itself provides additional details which are described in the following.

2.3.1 Memory Management and Communication Objects

The communication in terms of internal low level data transport in ACE/SmartSoft is mainly based on the class `SmartMessageBlock` which in turn is a direct mapping on the `ACE_Message_Block` class from the ACE library. The main function of this class is to store a marshalled stream buffer. The communication between components in `SmartSoft` is done by using communication objects. These objects provide both the platform dependent data

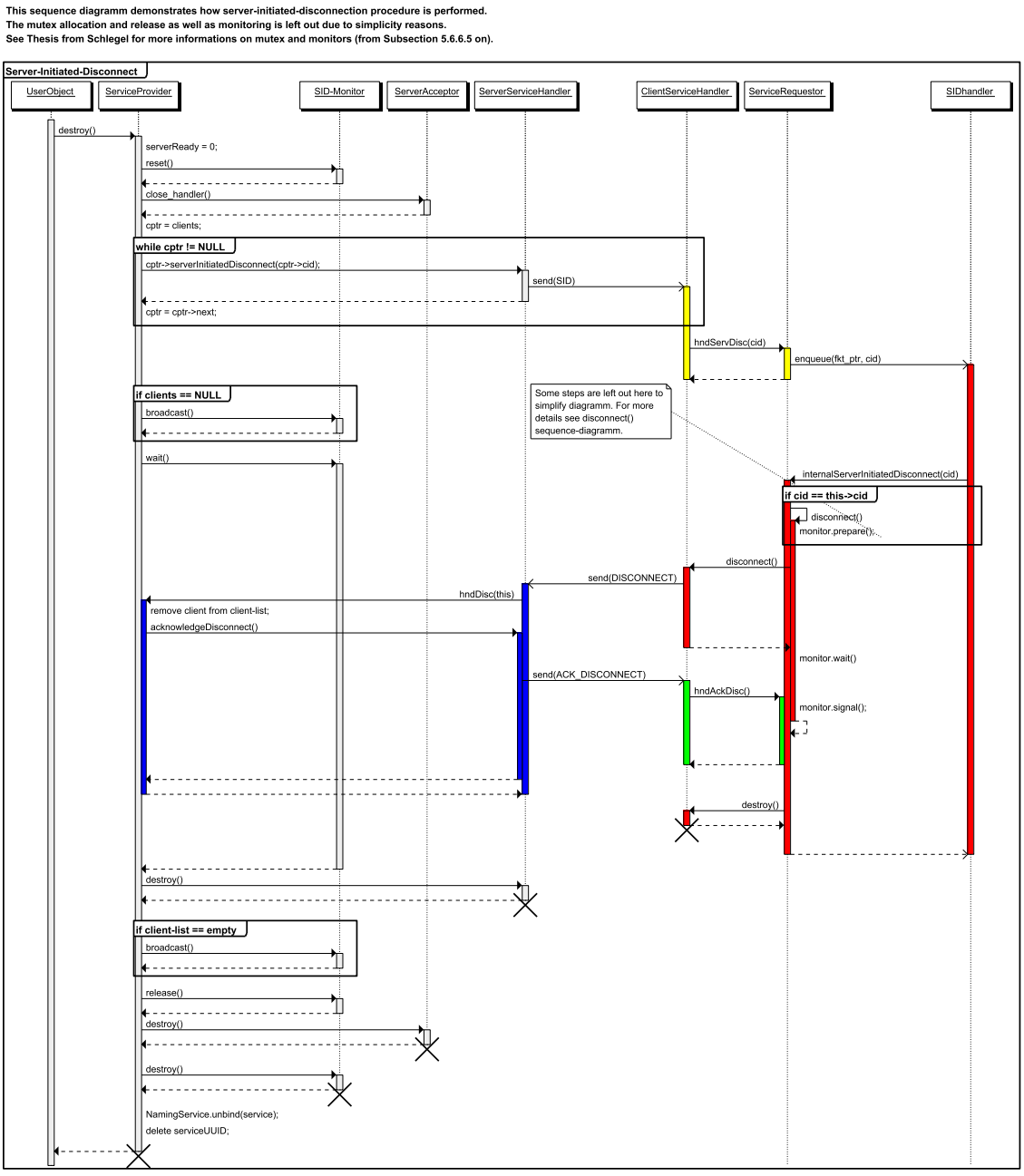

Figure 2.6: Sequence diagram: Destruction of a Service Provider with SID.

representation and the marshalling methods to transform this data into a platform independent representation. For this reason each communication object must consist of the methods `get` and `set`. The `get` method serialises the internal data of the communication object and stores it in a local instance of a `SmartMessageBlock` class. This method does not modify the internal data in the communication object, but makes a snap-shot copy of it. At this point a pitfall can occur. The copy of the data is typically stored on the heap and the pointer to this copy is passed on through several levels of the communication stack. In fact, the data is passed from the user level (from communication objects) to the SmartSoft level (inside of a communication pattern) and finally to the communication level (in ACE). It is very important to clearly define where and when the data is used and where and when the data is not needed any more and can be cleaned up. A not clear separation of responsibilities to create and destroy this data can lead either to a memory leak or to a segmentation fault.

The solution for this problem can be described by means of the following two scenarios.

Sending a Message

The solution for the memory handling problem as described above can be discussed using the `send` method from `SendClient` communication pattern. The sending functionality in other communication patterns work accordingly. The method `send` gets a reference to the communication object whose internal data has to be transferred to the connected `SendServer`.

Inside of the `send` method a copy of the internal data from the communication object is fetched by calling the `get` method of the communication object. The data is then stored in a local instance of type `SmartMessageBlock`. A pointer to this instance is forwarded to the `command` method of the `SendClientServiceHandler`, which in turn forwards the request to the `send_command_message` method of its base class `ServiceHandler`. The `ServiceHandler` sends the data from the `SmartMessageBlock` reference on its internal socket connection and returns with a success or an error code.

At this point the local instance of `SmartMessageBlock` in the `send` method from `SendClient` is not needed any more and must be cleaned up actively. Otherwise a memory leak will occur. Thus, a rule of thumb is that the instance of `SmartMessageBlock` must be cleaned up at the place where it was originally created. This is typically the place where the `get` Method of the communication object is called.

Receiving a Message

The memory handling during the reception of a message can be explained with the `SendServer` communication pattern. The message reception is triggered from the `handle_input` method of the internal `ServiceHandler` class. There the data from the socket is received and is stored inside of a local instance of a `SmartMessageBlock`. A pointer to this instance is forwarded to the corresponding callback method `hndCmd` of the `SendServer`. There a local communication

object is created and is filled with data by using its `set` method. From here on the instance of `SmartMessageBlock` is not needed any more.

Again, the rule of thumb is the same as above that the instance of `SmartMessageBlock` must be cleaned up at the place where it was originally created. This time, this place is the `handle_input` method of the `ServiceHandler` class. To simplify the implementation a *smart pointer* can be used in both cases for sending and receiving messages.

2.3.2 ServiceHandler and Callbacks

As shown in [2, page 158], different messages are communicated internally in SmartSoft between service providers and service requestors. There are two different message types, the administrative messages and data messages. Administrative messages are used internally in SmartSoft to coordinate the internal states of a server provider and connected service requestors. For example a call of the `subscribe` method of a `PushNewestClient` results in an administrative message to the `PushNewestServer`. The other type of messages are data messages. These messages are used to transmit the marshalled content of a communication object. For example a call of the `put` method in a `PushNewestServer` (giving a reference of a communication object as a parameter) results in marshalling of the communication object and the transmission of its content in a data message to the `PushNewestClient`. The complete overview of all possible messages in ACE/SmartSoft are listed in the table 2.1 in the subsequent subsection.

One of the most important requirements in SmartSoft on the underlying communication is the interoperability. This means, that the low level data on communication level must be completely independent of any platforms (i.e operating system, byte order, architecture, etc.). Therefore all messages must be transformed into a platform independent representation before sending. After receiving a message on the receiver side the message must be transformed back into representation corresponding to the current platform. For this purpose ACE provides the *common data representation (short CDR)* stream classes. In particular the two classes `ACE_OutputCDR` and `ACE_InputCDR` are of interest. `ACE_OutputCDR` provides a marshalling mechanism to transform simple (platform specific) data types into internal platform independent representation. The internal data representation in both CDR stream classes is based on the class `ACE_Message_Block`. That is, `ACE_Message_Block` stores internally a buffer with a marshalled representation, which can be directly transmitted on an `ACE_Socket` for example. Finally, the CDR class `ACE_InputCDR` can use the data from a `ACE_Message_Block` and transform it back into a platform specific representation in terms of demarshalling.

These three classes are the basis for all communication between components in ACE/SmartSoft. The physical communication of the low level data on a TCP socket is encapsulated inside of the class `SmartServiceHandler` in ACE/SmartSoft. To simplify the implementation of `SmartNamingHelper` and to provide a simple distinction between admin-

istrative and data messages, this class internally implements a simple protocol, which uses `ACE_Message_Blocks` as a basis. This protocol is described in the following.

Message Header: Has always a length of 16 Bytes and is structured as follows

```
| ByteOrder |  CMD-ID  | Param-length | MSG-length |
   4 Bytes    4 Bytes      4 Bytes       4 Bytes
```

- The first 4 bytes are used by the communication middleware to decide whether to swap or not the message content depending on the locally used byte order.
- The second 4 bytes are used by communication patterns in ACE/SmartSoft to identify the type of the internal message. All allowed message types depending on communication patterns are defined in the enumeration `SmartCommand`.
- If one particular message needs to transfer additional administrative parameters, the next 4 bytes shows the length of the corresponding buffer. Otherwise the length is set to 0.
- The last 4 bytes stores the length of the marshalled representation from communication object. For administrative messages this length is set to null.

Administrative Parameters: All administrative parameters - which are used inside of `ACE/SmartSoft` and are needed to coordinate the service requestors with its service providers - are stored in this buffer.

Message Content: This buffer contains the platform independent content of a communication object.

The class `SmartServiceHandler` provides a bidirectional point-to-point communication. Thus, this class can be used on both sides as an endpoint.

For sending data this class provides the method `send_command_message(...)`. As parameter, this method gets the command identifier, an optional message block containing administrative parameters and an optional message block containing the content of a communication object. These three parameters are used to generate a corresponding message header and to transfer the message payload to the remote endpoint. Considering the figure 2.2 as sown at the beginning of this chapter, the `send_command_message(...)` method corresponds to the generic interface C of a service requestor and a service provider. This method is called from corresponding methods inside of the communication patterns. These methods are shown in table 2.1 which corresponds interface B in figure 2.2.

Further, `ServiceHandler` implements the method `handle_input` which receives a message from its internal socket according to the message header and delegates the contents to the

method `handle_incomming_message(...)`. The former method corresponds interface C of a service requestor and a service provider in figure 2.2. The latter method is a pure virtual method which must be implemented in derived classes. This is the main place where the mapping to the callback functions of individual communication patterns is implemented, which corresponds to interface B of a service requestor and a service provider in the figure.

2.3.3 Overview of all Messages in SmartSoft

Table 2.1 shows the complete set of messages (see [2, table 5.45 and 5.46]) which are used internally in corresponding communication patterns in their ACE/SmartSoft implementation. The two patterns `DynamicWiring` and `StatePattern` are not listed in the table, because they are both based on the `Query` pattern. Thus, these two patterns are completely independent of any communication middleware details.

Pattern	Receivable Messages
Send	*SmartSendServerInterface* (Service Provider)
	R0 StatusCode connect(int cid, ACE_Utils::UUID *serviceID)
	R1 StatusCode discard()
	R2 StatusCode disconnect()
	D StatusCode **command**(SmartMessageBlock *message)
	SmartSendClientInterface (Service Requestor)
	A0 StatusCode acknowledgmentConnect(int cid, int status)
	A2 StatusCode acknowledgmentDisconnect()
	R3 StatusCode serverInitiatedDisconnect(int cid)
Query	*SmartQueryServerInterface* (Service Provider)
	R0 StatusCode connect(int cid, ACE_Utils::UUID *serviceID)
	R1 StatusCode discard()
	R2 StatusCode disconnect()
	D StatusCode **request**(SmartMessageBlock *msg, int QueryID)
	SmartQueryClientInterface (Service Requestor)
	A0 StatusCode acknowledgmentConnect(int cid, int status)
	A2 StatusCode acknowledgmentDisconnect()
	R3 StatusCode serverInitiatedDisconnect(int cid)
	D StatusCode **answer**(SmartMessageBlock *msg, int QueryID)
PushNewest	*SmartPushNewestServerInterface* (Service Provider)
	R0 StatusCode connect(int cid, ACE_Utils::UUID *serviceID)
	R1 StatusCode discard()

Table 2.1: The messages used internally by the communication patterns

Internal messages in communication patterns (continued)	
Pattern	**Receivable Messages**
	R2 StatusCode disconnect()
	R4 StatusCode subscribe(int sid)
	R5 StatusCode unsubscribe()
	SmartPushNewestClientInterface (Service Requestor)
	A0 StatusCode acknowledgmentConnect(int cid, int status)
	A2 StatusCode acknowledgmentDisconnect()
	R3 StatusCode serverInitiatedDisconnect(int cid)
	D StatusCode **update**(const SmartMessageBlock*, int sid)
PushTimed	*SmartPushTimedServerInterface* (Service Provider)
	R0 StatusCode connect(int cid, ACE_Utils::UUID *serviceID)
	R1 StatusCode discard()
	R2 StatusCode disconnect()
	R4 StatusCode subscribe(int cycle, int sid)
	R5 StatusCode unsubscribe()
	R6 StatusCode getServerInformation()
	SmartPushTimedClientInterface (Service Requestor)
	A0 StatusCode acknowledgmentConnect(int cid, int status)
	A2 StatusCode acknowledgmentDisconnect()
	R3 StatusCode serverInitiatedDisconnect(int cid)
	A4 StatusCode acknowledgmentSubscribe(int active)
	A6 StatusCode serverInformation(double cycle, int active)
	R7 StatusCode activationState(int active)
	D StatusCode **update**(const SmartMessageBlock*, int sid)
Event	*SmartEventServerInterface* (Service Provider)
	R0 StatusCode connect(int cid, ACE_Utils::UUID *serviceID)
	R1 StatusCode discard()
	R2 StatusCode disconnect()
	R4 StatusCode activate(int mode, int aid, const SmartMessageBlock *params)
	R5 StatusCode deactivate(int aid)
	SmartEventClientInterface (Service Requestor)
	A0 StatusCode acknowledgmentConnect(int cid, int status)
	A2 StatusCode acknowledgmentDisconnect()
	R3 StatusCode serverInitiatedDisconnect(int cid)

Table 2.1: The messages used internally by the communication patterns

Internal messages in communication patterns (continued)	
Pattern	**Receivable Messages**
	A4 StatusCode acknowledgmentActivate(int status)
	D StatusCode **event**(CHS::SmartMessageBlock *user, int id)

Table 2.1: The messages used internally by the communication patterns

2.4 Further Details explained with the PushNewest pattern

The implementation of the mapping of communication patterns to the ACE communication middleware follows the same pattern which can be explained by using the PushNewest pattern as an example. For this purpose, first the class diagram is explained to show the relationships between single classes of the PushNewest pattern. After that, the relevant files are listed and explained to give a quick overview for a framework builder who wants to map this pattern to a different communication middleware (or searches for a particular part of the mapping).

2.4.1 Class Diagram

The internal structure with relevant classes for the mapping of the PushNewest pattern to the ACE communication middleware is illustrated in figure 2.7. This structure is identical for each other communication pattern, apart from some pattern specific, middleware independent parts (like the `QueryServerHandler` for example).

The two interface classes *SmartPushNewestClientInterface* and *SmartPushNewestServerInterface* represent so called *interface objects*. They define all legal message calls which result in corresponding messages on the communication channel. Thus, these interfaces define the messages which can be internally transmitted from the PushNewestClient to the PushNewestServer and vice versa. An overview of all messages for all communication patterns is shown in table 2.1 above.

The two classes *PushNewestClient* and *PushNewestServer* define the interface which can be used by component developers and which is completely independent of any communication middleware details. The implementation of these classes provide the core logic of the PushNewest pattern, which is independent of the communication middleware. This is done by delegating message calls to corresponding service handlers and by providing callback methods to receive messages from the service handlers.

The class *SmartServiceHandler* provides the method `send_command_message` to send messages on the communication channel. For this purpose this class has a direct access to its internal TCP socket, which is connected to a corresponding remote service handler. This

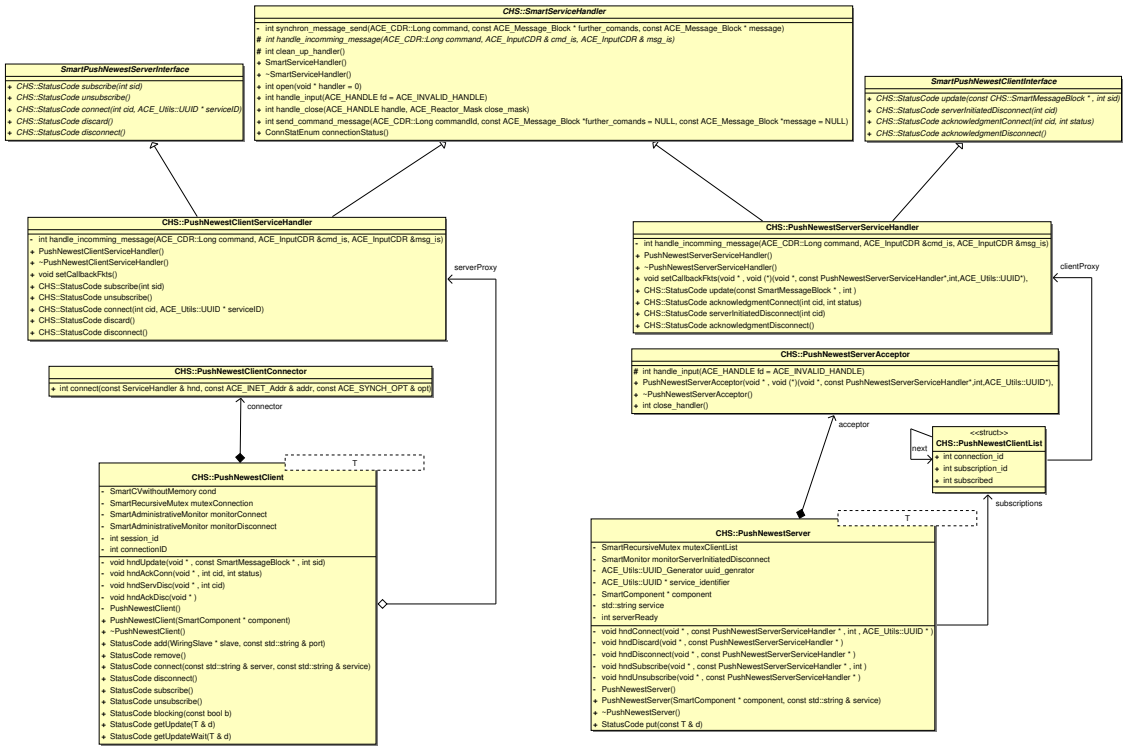

Figure 2.7: Class diagram overview of the PushNewest pattern

method is used for both, to transfer administrative messages which coordinate the internal states of the PushNewestClient and PushNewestServer, and the data payload from communication objects which is transmitted in this case from PushNewestServer to PushNewestClient. The method `send_command_message` internally calls the method `synchron_message_send` which in turn processes the real transmission taking the *delivered* policy into account.

Each TCP socket is controlled out of the Reactor in ACE. On arrival of a message on the socket an input-event is fired which is handled by the Reactor by calling the `handle_input` method of a corresponding service handler. Inside of the service handler the incoming data stream is received and demarshalled according to the header of the message. This step is independent of the message content and is always the same for all communication patterns. In the next step the message is processed individually for each communication pattern. Therefore the pattern specific method `handle_incomming_message` is called. Inside of this method a dispatching is performed where one of the callback methods of the corresponding communication pattern is called. For the dispatching the *command-id* is used (see corresponding enumeration in Doxygen[1]).

The whole procedure for a service provider is shown in the following by means of a *Push-*

[1] `http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/namespace_c_h_s.php`

NewestPattern. A *connect* call from a PushNewestClient to the PushNewestServer results in a call of the `handle_input` of the `PushNewestServerAcceptor`. This method internally creates a new instance of a `PushNewestServerServiceHandler` which is the individual endpoint for the connection. To be able to register this `PushNewestServerServiceHandler` in the `Reactor`, `PushNewestServerServiceHandler` is derived from the `SmartServiceHandler` class. During the initialisation of the `PushNewestServerServiceHandler`, the function pointers of the static upcalls in the superodinate communication pattern are initialized. For this purpose the `PushNewestServerAcceptor` internally contains these pointers. For the PushNewestServer pattern there are the following pointers available: `hndConnect`, `hndDiscard`, `hndDisconnect`, `hndSubscribe` and `hndUnsubscribe`. Finally, the `PushNewestServerServiceHandler` is activated such that it is ready to receive/send messages. The connection is finished first if a corresponding connection message arrives in the `PushNewestServerServiceHandler` and the service identifier (inherited in this message) matches the local service identifier in the PushNewestServer pattern. In this case an acknowledgement connect message is replied.

For a service requestor this procedure is shown by means of the *PushNewestClientPattern*. The method `connect` of this class uses the `PushNewestClientConnector` to create and connect a new instance of a `PushNewestClientServiceHandler`. This represents the physical connection to the PushNewestServer. After that the connection message is transmitted to the PushNewestServer and the connection is successfull if an accept-connection message replies. According to the service provider, the following upcall messages are available in PushNewestClient: `hndUpdate`, `hndAckConn`, `hndServDisc` and `hndAckDisc`.

2.4.2 File structure

The file structure for each communication pattern follows always the same pattern. Therefore the structure can be explained using the PushNewest communication pattern as an example.

- `smartPushNewest.hh`
  A component developer in the role of a framework user must include this file to use the PushNewest communication pattern.

- `smartPushNewest.th`
  This file contains the implementation of the templates from the PushNewest communication pattern. This file is only to modify by the framework builder. In fact this file implements the core logic which is independent of the underlying operating system and communication middleware. This logic can be used as a basis for mappings of this pattern to different communication middlewares.

- `smartPushNewestPattern.hh`,
  `smartPushNewestServerPattern.hh`, `smartPushNewestClientPattern.hh`,

`smartPushNewestServerPattern.cc`, `smartPushNewestClientPattern.cc`
The interface objects defined in the first file represent the two interfaces (client and server) containing all messages which PushNewest pattern is able process. The subsequent two header files contain the definition for the implementation of these interfaces and the service handler objects. Additionally the `smartPushNewestServerPattern.hh` file contains the definition of the `Acceptor` for PushNewestServer. `smartPushNewestClientPattern.hh` file contains the definition of the `Connector` for PushNewestClient. The last two files contains the implementation of the service handler objects as well as the `Acceptor` (and resp. `Connector`). A framework builder - who wants to map this communication pattern to one other communication middleware - must modify the mapping in these files. For example the method `handle_incomming_message` - which process the mapping from a general message coming from the `handle_input` method to the callback functions of the pattern - is also implemented there.

- `smartServiceHandler.hh`, `smartServiceHandler.cc`
  This file contains the pattern independent glue logic for the mapping to the underlying communication middleware. This logic contains of a generic implementation of the point-to-point communication channel called `SmartServiceHandler`. The pattern independent part to receive messages is implemented in the `handle_input` method and to send messages is implemented in the `send_command_message(...)` method. These files are also only to modify by a framework builder, who wants to map the communication object to a different communication middleware.

Chapter 3

The SmartSoft Kernel

3.1 The Number of Threads inside a Component

The SmartSoft framework allows a component developer to use an arbitrary number of threads on the user level. ACE/SmartSoft is completely thread safe, thus all communication patterns and the `SmartComponent` class can be used out of arbitrary threads at the same time. Besides the user threads, a `SmartComponent` requires a set of threads to handle its internal activities. These threads are described in the following.

In principle there are two options for how many threads are at least needed inside a component in ACE/SmartSoft. Both options are suitable solutions for the mapping of SmartSoft to the ACE communication middleware and have their pros and cons. Both options (labeled A and B) are described in detail in the following sections. The current version 1.7.2 of ACE/SmartSoft implements option B for efficiency reasons as will be explained in detail.

3.1.1 The minimum number of Threads

Independently of the option A or B, each component requires a minimum number of threads for its internal activities. The class `SmartComponent` in ACE/SmartSoft operates the communication of a component by grasping the thread which called its `run` method. Besides grasping that calling thread, it starts three more threads:

Main Thread: The `run` method of a component uses the calling thread to run its main loop. This thread is provided from outside. In most cases, the `run` method is called from the `main` program. The main loop handles all the communication activities of the component.

TimerThread: Each component implements a timer, which is used to handle all timings of a component. The timing service is for example used by the push timed communication pattern. The `PushTimedServer` gets regularly triggered to provide updates to

subscribed clients. A component wide timer is much more reasonable than having individual timers in each push timed server instance, for example. In the future, the timer will also be used to implement timeouts at the user API of the communication patterns. The timer needs its own thread to ensure that it can trigger activities independently of the communication activities.

SIDhandler: The *Server Initiated Disconnect Procedure* as described in section 5.6.6.10 of [2] requires an active queue. The active queue is set into operation within the `run` method of `SmartComponent`. However, as long as no server initiated disconnect activities are being invoked, this thread is just blocked and pending.

ShutdownTask: The shutdown of a component requires an independent thread to coordinate the shutdown process. This thread is openend when the shutdown of a component is invoked. At runtime of a component, this thread is not openend and thus does not require any resources. The main functionality of the `ShutdownTask` is to process the shutdown procedure, to observe the clean-up of resources and to ensure that a component is closed in all cases, even if some of the tasks do not shutdown cooperatively (see section 3.3).

Details of the mapping of the *connection oriented split protocol* are explained on pages 197-199 of [2]. In the most general case, a service requestor possesses two interface objects, namely the one for incoming messages and the one for outgoing messages. A service provider possesses one interface object for incoming messages and an interface object for each connected client.

The structure of the interface objects within the mapping of the communication patterns on ACE/SmartSoft has already been illustrated in figure 2.2. The ACE/SmartSoft implementation of the interface objects differs from the most generic implementation illustrated in figure 5.105 of [2] as follows:

- instead of having separate interface objects for incoming and outgoing messages, the ACE/SmartSoft implementation of a service requestor just uses a single interface object that handles all incoming and outgoing messages.
- instead of having one interface object for all incoming messages and one interface object per connected client for the outgoing messages, we again have combined interface objects. In case of the service provider, there is an interface object per connected client that handles all incoming and outgoing messages for that particular client.

Merging the interface objects for incoming and outgoing messages as described above is reasonable since the ACE service handlers support two-way communication. As explained in section 5.6.6.11 of [2], the interface object for incoming messages of all service requestors

can share one thread. In contrast, each interface object for incoming messages at a service provider requires its own thread. Since the ACE/SmartSoft implementation combines the interface objects for incoming and outgoing messages, the set of required threads needs to be chosen carefully. The two different kinds of mapping are described as option A and option B.

3.1.2 Option A

Each component possesses only a single reactor that dispatches the incoming messages to the appropriate service handlers. A service handler represents the interface object of a particular service requestor or service provider. The reactor of a component is run by the thread that called the `run` method of `SmartComponent`.

ACE provides the class `ACE_Svc_Handler` which is used to implement the interface objects in ACE/SmartSoft. In its default implementation, this class is passive. Its handler methods for incoming messages are triggered by the `reactor` at which the `ACE_Svc_Handler` is registered. The `ACE_Svc_Handler` can be parameterized and implemented as an active object that uses its own thread of control as shown in [1, page 215]. That already comes close to the requirement explained in section 5.6.6.11 of [2].

However, there is still a single reactor handling all incoming messages for service providers. Therefore, it might now happen that the single reactor is not able to forward incoming messages for a service provider in case the according `ACE_Svc_Handler` cannot accept the incoming message. That might happen because a tailback of messages for that particular service provider does not allow the reactor to forward the message to the interface object. In that situation, the reactor gets blocked as well and thus, even messages for other service providers get not forwarded anymore due to the blocked reactor.

To overcome that situation, the overall correct implementation would be to have individual reactors per interface object at service providers. At a service provider, only one service handler is registered at the reactor and thus no decoupling of the reactor and the interface object is required. Therefore, the `ACE_Svc_Handler` can be passive with a thread per reactor.

All interface objects of service requestors can share a single reactor (that is operated by a thread) and all registered `ACE_Svc_Handler` can be passive. At service requestors, it is guaranteed that all upcalls into the communication patterns can forward their messages without blocking. Thus, a single thread of the shared reactor cannot be blocked and thus can be shared.

However, this most generic solution would require one reactor for all service requestors of a component and a reactor per service provider of a component. Each reactor requires a thread to be operated. Of course, since only a single service provider is attached to the reactor, this thread could also be used via the upcalls to the SmartSoft communication pattern to operate the user level handler to process incoming requests. However, as soon as we need a different threading model at the user level (using `SmartProcessingPatterns`, e.g.

a thread pool), the upcalling thread is not fully utilized anymore.

3.1.3 Option B

This option uses a single reactor per component that operates all interface objects. However, we now have to ensure that the reactor never gets blocked by not being able to forward incoming messages to appropriate `ACE_Svc_Handlers`. The only critical upcalls from the reactor via a `ACE_Svc_Handler` into a communication pattern are user level handlers at service providers. User level handlers are only provided with the `SendServer` and the `QueryServer`.

With option B, passive user level handlers of service providers are not allowed to block on resources that depend on communication. For example, a passive handler is not allowed to wait for a resource which gets released only after further communication took place. Of course, it is also not allowed to invoke further communication from within a passive handler. The only allowed communication from within a passive handler is to call the `answer` method of the `query` service provider.

With active user level handlers, any kind of blocking is not a problem as long as the active user level handler accepts incoming requests for himself. The user level handler can enqueue incoming requests and in case of too many open requests reject processing them by answering with appropriate states in the answer to the service requestor.

That overall scheme of when to use active handlers and when to use passive handlers for processing requests at service providers is already recommended good practice and is thus compatible to the already established use of the communication patterns.

The mapping of option B is implemented in ACE as follows. The base class `ACE_Svc_Handler` can be used for both, the service requestors and the service providers in the same way. In the current implementation of ACE/SmartSoft this class is derived in the `SmartServiceHandler` class, which is parameterized to be both, a sender and a receiver at the same time. This class can be registered in the `Reactor` and a component can use the same `Reactor` for all communication patterns. Thus, the component needs only one main thread (plus the internal TimerThread and SIDhandler thread as described above).

Although the component developer must use the two handlers `SendServerHandler` and `QueryServerHandler` with more care, he gets also more freedom to optimize the efficiency of his component. The number of threads can be reduced to the very minimum. A nice side effect is that the implementation of the communication patterns in ACE/SmartSoft is simplified.

Summing up, both approaches – Option A and B – have their pros and cons. Currently the option B is implemented in the ACE/SmartSoft version 1.7.2. However, the option A can be implemented accordingly, because the ACE library allows both versions.

3.2 SmartComponent

`SmartComponent` is the main class of each component in SmartSoft. It can be seen as a component hull which implements the core functionality required in every component in SmartSoft. During the creation of a component, it initializes several resources (like internal threads as described in the previous section) which are used for example by communication patterns. More details for the initialisation procedure are shown in the first part in the following. A further important aspect of each component is its destruction (resp. shutdown) procedure, which is described in the second part below.

3.2.1 SmartComponent Initialization

The initialization procedure of a `SmartComponent` in ACE/SmartSoft is shown in the sequence diagram in figure 3.1.

The main focus during the initialization of a component in SmartSoft is to initialize its internal resources. In ACE/SmartSoft there are the following fife resources: `SIDhandler`, `ShutdownTimer`, `TimerThread`, `NamingHelper` and the `Reactor`. The meaning of the threads `SIDhandler`, `ShutdownTimer` and `TimerThread` is described in the foregoing section 3.1. The initialisation of the `NamingHelper` and the `Reactor` provides further details, which are described in the following.

As introduced in chapter 1 the naming service has a provider and a requestor sides. The provider is implemented as the naming service daemon. The daemon provides a centralised directory service. Therefore the daemon must be started before any of the components can be initialised. The requestor is implemented as the `NamingHelper` class in ACE/SmartSoft. This class is used by each component and the communication patterns in particular. In the current version 1.7.2 of `ACE/SmartSoft` this class is implemented as a singleton. This has the consequence that a separate instance of this class is initialized for each component in case that each component is initialised in a separate process (or even separate machine in the network). For the case where several components are initialised in the same process (in separate threads) only one instance of `NamingHelper` is initialized by the component which accesses this class at first. The other components in the same process share the same instance of the `NamingHelper` class.

This is convenient for the following reasons. An arbitrary number of naming service requestors can be connected to the naming service daemon and the daemon can handle an arbitrary number of requests without causing a deadlock. All requests coming from different requestors are serialized in the naming service daemon. The consequence is that if several requestors query for key/value entries in the naming service daemon at the same time, some of these requestors must wait before other requests are finished. This is not critical in ACE/SmartSoft, because any requests occur only in the initialization of service providers or in

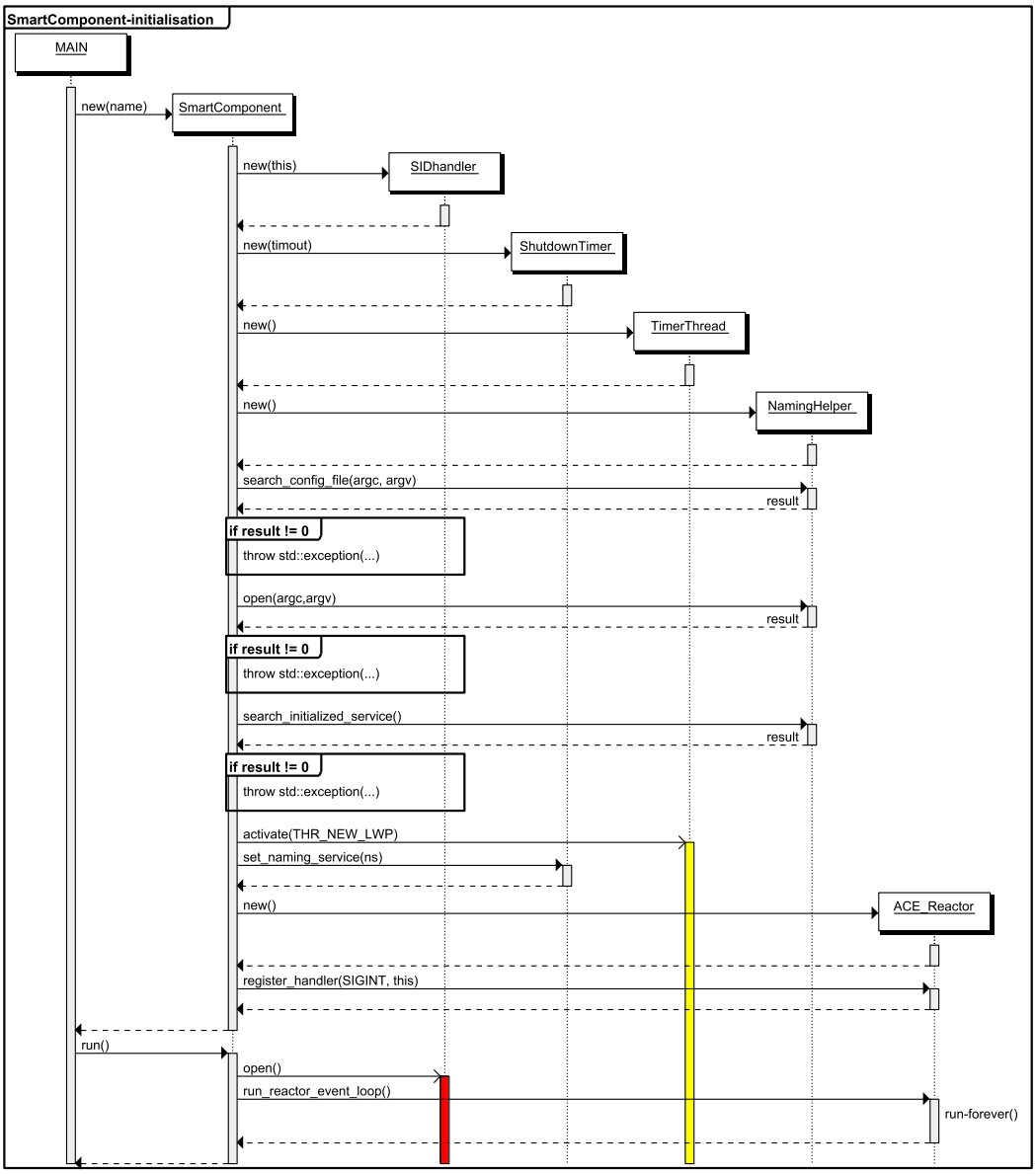

Figure 3.1: Sequence diagram: Initialization of a component.

the connection routine of service requestors. For example a connection routine is defined in SmartSoft such, that it can take up to one second to complete the connection. This time is capable in regular cases, otherwise a timeout return value indicates a connection problem in the corresponding service provider, which is a regular (and uncritical) behavior. Thus, a little delay for naming service requests does not violate any critical timings inside of components.

The `NamingHelper` class is based internally on a *Service Configurator Framework* of ACE as described in [1, chapter 5]. This feature provides the possibility to initialize a dynamic service – in this case it is the naming service requestor in the form of a `NamingHelper` – by using a configuration file for to parametrize this service. This allows to use a component with different configurations (the naming service and the components can be started on arbitrary machines) without recompilation. Of course, if the component must be executed on a machine with a different architecture to that where the components was compiled for, this component must be recompiled.

Additionally thereto the class `NamingHelper` tries internally to connect to the remote naming service daemon and fails if either the daemon is not running or is not reachable on the TCP address as defined in the configuration file. The consequence of this is, that a successful initialization of a `SmartComponent` in ACE/SmartSoft is only successful if the following conditions are met:

1. The configuration file for the component is available and the component is able to access this file (either using the default location – which is the local folder – or the $-f$ console parameter)
2. The naming service daemon is not running or is not reachable with the TCP address that is given in the configuration file.
3. The internal connection procedure of the `NamingHelper` is successful and the service is started.

A violation of one of these conditions result in a corresponding error message on the console and an abortion of the component's initialisation procedure. A component is not started in this case. Otherwise the component proceeds with the opening of the `TimerThread`.

After that, the `Reactor` of this component is initialised. Again, the Reactor needs to be started separately for each component, as described in the foregoing section. Finally, the component registers its handler to react on the `SIGINT` signal, which is fired for example with the keys `Ctrl+C`. The component is now fully initialised, but is not actively running. The component runs first, after the call of its `run` method.

3.2.2 Shutdown Procedure of SmartComponent

The shutdown procedure of a `SmartComponent` in ACE/SmartSoft is shown in the sequence diagram in figure 3.2. As described in the foregoing subsection the component reacts on the

`SIGINT` signal (resp. `Ctrl+C` key combination). In fact, the `SIGINT` signal is cached by the `Reactor` of the component. The `Reactor`, on the other hand, calls the internal `handle_signal` method of the `SmartComponent`, which in turn initialises the shutdown procedure.

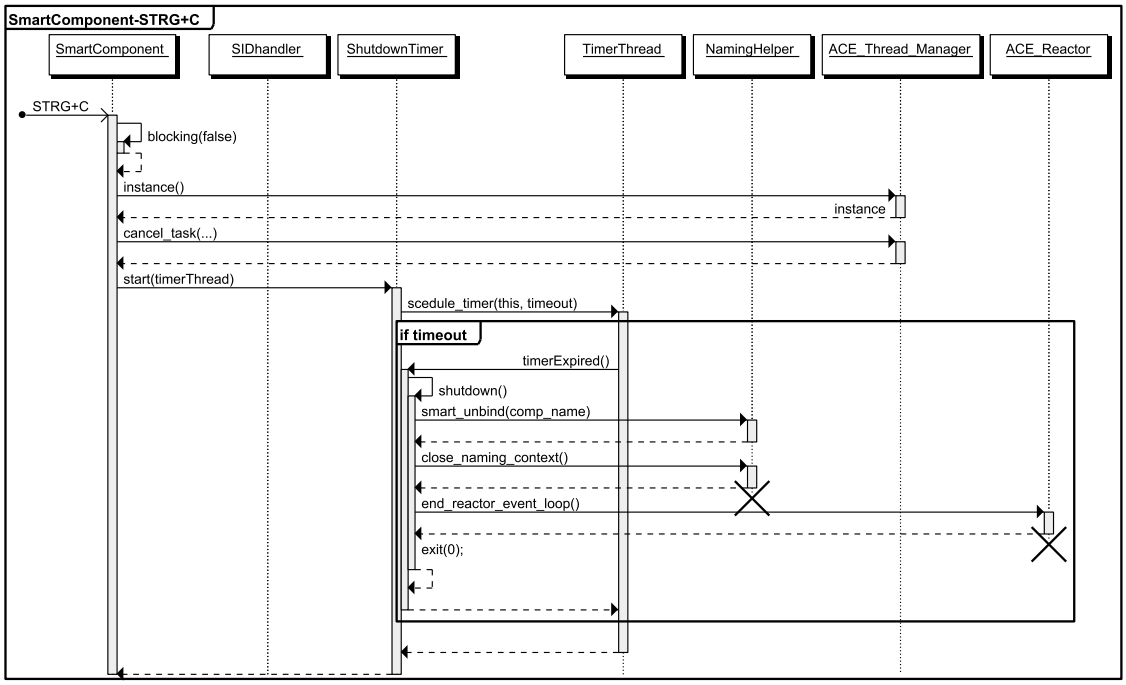

Figure 3.2: Sequence diagram: Handling of ctrl+c by a component.

The whole shutdown procedure is completely independent of the underlying communication middleware. However, the thread management is based on the platform independent threading features provided in the ACE library. The procedure consists of the following steps:

- First, the component switches its internal *blocking* state to false. This is a powerful feature in SmartSoft to unblock all attached communication patterns such that all `ManagedTasks` which are blocked on communication are able to shut down. The consequence is, that all blocked service requestors and service providers unblock immediately with a corresponding status code.

- After that, the `ManagedTasks` are signalled to shut-down. Therefore the so called *cooperative cancellation* is performed. This is done using the `ACE_Thread_Manager`, which runs as a singleton in the process. The corresponding `ManagedTasks` recognise the cancel signal and end up their thread main loop. This is described in more detail in 3.3.

- Immediately after signaling the threads, the `ShutdownTimer` is started. This is performed by calling its `start` method and the reference to the component's `TimerThread` is passed as parameter.

- The `start` method of the `ShutdownTimer` consists of the following steps. First, the reference to the `TimerThread` is used to initialise a one-shot timer. With that, the time is specified, how long the `ShutdownTimer` waits before shutting down the component. This time is necessary to give the `ManagedTasks` a chance to clean up their resources and to close their threads.

- If all `ManagedTasks` closes in time, the `ShutdownTimer` deactivates the timer and shuts the component down (immediately after the last of `ManagedTasks` is closed).

- If one (or some) of the `ManagedTasks` do not close in time (for example because the task blocks on a uncontrolled resource), a timeout occur and the `timer_expired` method of `ShutdownTimer` is called. In this case the `ShutdownTimer` do not wait any longer and shuts down the component immediately. Thereby, all entries in the naming service which are related to the service providers of this component are cleaned up by calling the `smart_unbind` method of the `NamingHelper` class. After that, the `NamingHelper` and the `Reactor's` main loop are closed. This procedure is illustrated in the sequence diagram in figure 3.2.

3.2.3 Administrative Monitor in SmartComponent

An overview of all classes – which directly interact with the `SmartComponent` class – are shown in the class diagram in figure 3.3.

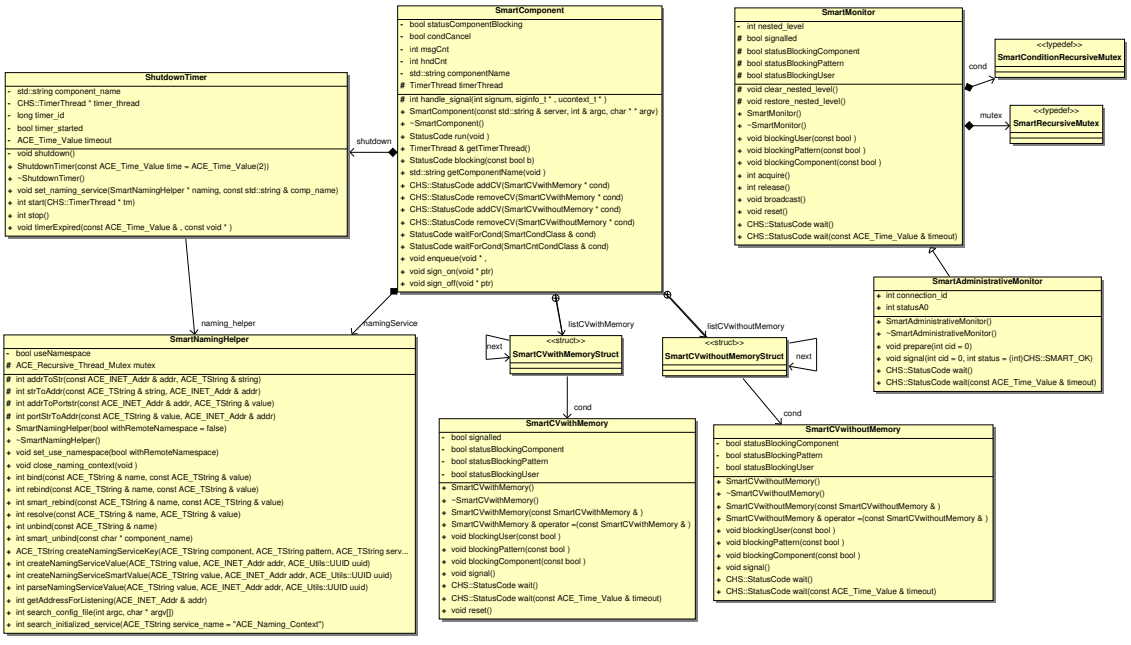

Figure 3.3: Class diagram: Overview of classes directly related to SmartComponent.

The ShutdownTimer and the NamingHelper (in figure 3.3 on the left) are described in the foregoing subsection. A description of all methods is online available in Doxygen[1]. The class SmartComponent uses internally the wrappers SmartCVwithMemory and SmartCVwithoutMemory implement blocking calls inside of communication patterns. These two classes simple wraps the functionality of the class ACE_Condition_Thread_Mutex, available in the ACE library.

Additionally to the two classes, the Monitor class and the AdministrativeMonitor class in particular are used inside of communication patterns to implement the connection and disconnection procedures. These two classes implement the specification defined in [2, pages 170-174]. As a basis the two classes SmartConditionRecursiveMutex and SmartRecursiveMutex are used. The SmartConditionRecursiveMutex is a mapping on the class ACE_Condition_Recursive_Thread_Mutex and SmartRecursiveMutex directly maps on the class ACE_Recursive_Thread_Mutex, which are both available in the ACE library.

3.3 ManagedTasks

One of the core features in the ACE/SmartSoft implementation is the ManagedTask class. This class derives from the ACE_Task class in the ACE library and enhances it by the feature to manage this task out of the corresponding component. This management affects the creation and the destruction of ManagedTasks. In the following the current implementation of the ManagedTask is described which can be found in the ACE/SmartSoft version 1.7.2.

The core functionality of the ManagedTasks is to provide a unified and platform independent possibility to implement multithreading applications (resp. components) for a component developer. For this purpose the ManagedTask internally parametrises its base class such that one LWP[2] thread is started by calling the start method of ManagedTask. According thereto, the ManagedTask can be stopped by calling its stop method.

The methods on_entry, on_execute and on_exit can be used to implement the logic, which is intended to be executed out from the internal thread inside of the ManagedTask. In brief, the on_entry method is called (out of this thread) once after the thread is started and can be used to implement the initialisation of resources which might be used in the thread. The on_execute method is called from an infinite loop (out of the thread) till the ManagedTask is signalled to close (see below). The on_exit method is called once after the internal thread loop is left and is the last method that is called before the thread exits. This method is meant to clean up resources that are used in the thread (for example close files, etc.).

On the first view the class ManagedTask is independent of the SmartComponent and can be used as is without limitations. However, if using this class together with a SmartComponent it

[1] http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/namespace_c_h_s.php
[2] Light-Weight-Process

provides one additional feature. If a pointer to a `SmartComponent` is passed as an parameter in the constructor of the `ManagedTask`, this task becomes managed by this component. This means, that if the component shuts down, it can prior to that signal this task to close down and the component waits till the task is down (or a timeout occur – see previous section).

The closure of a `ManagedTask` from a `SmartComponent` works as follows. As shown in figure 3.2 the component uses the `ACE_Thread_Manager` to signal tasks to close by calling the `cancel_task` method. For the parameter argument of this method, the component uses an internal instance of type `ACE_Task`. A pointer of to this instance is passed as parameter in the `cancel_task` method. This has the following reason. The class `ACE_Thread_Manager` needs an instance of type `ACE_Task`, which can be used as the parent object for other `ACE_Task` instances. In other words a set of `ACE_Tasks` can be associated with one particular instance of `ACE_Tasks` (the base task). This has the advantage, that now a call of `cancel_task`, giving the pointer to the base task, closes all associated tasks also. This base task represents an empty hull and its internal thread must not be running for this feature. Thus, the base task does not use additional (unnecessary) resources.

```
1  int ManagedTask::start()
2  {
3    ACE_GUARD_RETURN(CHS::SmartRecursiveMutex, guard, mutex, -1);
4
5    if(!thread_started)
6    {
7      int retval = this->activate
8          (
9            THR_NEW_LWP  // initialize as kernel-level thread
10           ,1           // initialize exactly one thread
11           ,0           // do not force to activate if already activated earlier
12           ,ACE_DEFAULT_THREAD_PRIORITY // default priority
13           ,-1          // group id is chosen automatically
14           ,baseTask    // set base class to the dummy
15           ,0,0,0       // ignore thread_handles and thread-stack
16           ,thread_ids  // save this thread-id (to be able to close this thread)
17         );
18     if(retval == 0) {
19       this->open();
20     }else{
21       return -1;
22     }
23   }
24
25   return 0;
26 }
```

Listing 3.1: Internal implementation of the Start method

The feature described above is used in ACE/SmartSoft as follows. A `SmartComponent` initialises internally an instance of `ACE_Task` as the base task dummy. This instance is used each time when a `ManagedTask` is started (as shown in line 14 of listing 3.1). If the component now closes, the base task dummy is passed to the method `cancel_task`. Thus, each `ManagedTask` is signalled to close, because it is associated with the base task dummy.

```
1  int ManagedTask::svc()
2  {
3    bool stop = false;
4
5    if(this->on_entry() != 0) stop = true;
6
7    while(!test_cancel() && !stop)
8    {
9      if(this->on_execute() != 0) stop = true;
10   }
11
12   return this->on_exit();
13 }
```

Listing 3.2: Internal implementation of the svc method

The closure signal is detected in a `ManagedTask` by calling the `test_cancel` method from within the `svc` method inside of the task. This is illustrated in line 7 of listing 3.2. The `test_cancel` method is provided by the `ACE_Task` class.

Summing up, the `ManagedTask` provides a platform independent task, whose cancellation can be additionally managed by its superordinate component.

Chapter 4

The User View

The user view on ACE/SmartSoft comprises two different facets. The first one is of interest to the administrator of ACE/SmartSoft who has to install the framework and who has to manage the upgrades etc. The second one is related to the robotics user of ACE/SmartSoft who wants to build and reuse components based on ACE/SmartSoft.

4.1 The Administrator View

- ACE/SmartSoft is hosted on sourceforge and can be reached via `http://smart-robotics.sourceforge.net/aceSmartSoft/`
- the ACE/SmartSoft repository can be reached via `http://sourceforge.net/projects/smartsoft-ace/develop`
- installation instructions can be found on the home page of ACE/SmartSoft on Sourceforge
- the ACE/SmartSoft core library contains a string identifying the release number of the library. The ACE/SmartSoft release number consists of three entries according to the standard scheme for software releases. Beginning with version number 1.7.2, the following scheme is used:
  - Version number 1.7.2 indicates the major release number 1, the minor release number 7 and the revision number 2.
  - Revision numbers just increment with patch releases, cleaning up source code etc.
  - Minor release numbers increment when new features and new functionality has been added.
  - Major release numbers change when the overall structure has undergone modifications.

  - All releases with the same major and minor number are fully interoperable.

4.2 The Robotics User View

- An extensive *User Guide* on ACE/SmartSoft can be found on `http://smart-robotics.sourceforge.net/aceSmartSoft/userguide.php`
  - it describes the various communication patterns and their usage
  - it comprises various step-by-step instructions that cover the most often needed solutions
    * How to build a communication object?
    * How to build a SmartSoft/ACE component?
    * A practical SmartSoft/ACE component example
    * How to port a SmartSoft/CORBA component?
- Details on the communication patterns are also documented by a Doxygen documentation generated out of the source code of ACE/SmartSoft. That documentation can be found on `http://smart-robotics.sourceforge.net/aceSmartSoft/doxygen/`
- All generic documentation for the CORBA/SmartSoft release of SmartSoft also applies to the *ACE/SmartSoft* release. The CORBA/SmartSoft release can be found on `http://smart-robotics.sourceforge.net/corbaSmartSoft/`.

Bibliography

[1] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] C. Schlegel. Navigation and execution for mobile robots in dynamic environments: An integrated approach. *PhD thesis, University of Ulm*, 2004.

[3] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

Berichte des ZAFH Servicerobotik
ISSN 1868-3452

Herausgeber:
ZAFH Servicerobotik
Hochschule Ulm
D-89075 Ulm

http://www.zafh-servicerobotik.de/

Investition in Ihre Zukunft
gefördert durch die Europäische Union Europäischer Fonds
für regionale Entwicklung
und das Land
Baden-Württemberg