INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung systemnaher Adaptoren für Datenquellen auf Linux-Systemen

Nico Weiland

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer
Nils gentschen Felde

Abgabetermin: 26. September 2007

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Entwicklung systemnaher Adaptoren für Datenquellen auf Linux-Systemen

Nico Weiland

Aufgabensteller: Prof. Dr. Heinz-Gerd Hegering

Betreuer: Vitalian Danciu
Martin Sailer
Nils gentschen Felde

Abgabetermin: 26. September 2007

Hiermit versichere ich, dass ich das vorliegende Fortgeschrittenenpraktikum selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 26. September 2007

. . . . . .

(Unterschrift des Kandidaten)

In den letzten Jahren hat das Dienstmanagement gegenüber dem Netz- und Systemmanagement immer weiter an Bedeutung gewonnen. Das Ziel ist die Schaffung einer dienstorientierten Sicht, so dass ein Dienst als Einheit betrachtet und als solche verwaltet werden kann. Die Eigenschaften der Dienste, die sogenannten Dienstattribute, setzen sich dabei aus den Attributen der Netz- und Systemkomponenten zusammen. Um diese Attribute auszulesen und zu Dienstattributen zu aggriegieren, die dann an die Mangementanwendung weitergegeben werden können, wurde am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung eine Schichtenarchitektur für das Dienstmanagement entwickelt.

In diesem Fortgeschrittenenpraktikum sollen systemnahe Adaptoren implementiert werden, die eine Schicht in dieser Architektur darstellen und dafür zuständig sind, auf Datenquellen zuzugreifen und eine Normalisierung des Formats der erhaltenen Daten durchzuführen. Dieser Zugriff soll sowohl indirekt über die Dienste einer unteren Schicht möglich sein, als auch direkt über systemnahe Operationen. Weitere auszuübende Funktionen der Adaptoren sind zum Beispiel die Überprüfung, ob die Daten gewissen Kriterien genügen oder ein konstanter Zugriff auf die Datenquellen gemäß einem bestimmten Intervall. Schließlich müssen die Informationen einer höheren Schicht zur Verfügung gestellt werden, die daraus dann die Dienstattribute erstellen kann. Zentrale Aufgabe bei der Entwicklung der Adaptoren ist die Spezifikation einer geeigneten Schnittstelle, über die die Schichten miteinander kommunizieren können. Des weiteren sollen die Adaptoren sowohl das Push-Konzept, als auch das Pull-Konzept unterstützen. Außerdem wurde großer Wert auf einfache Erweiterbarkeit gelegt, so dass sich neue Datenquellen leicht integrieren lassen und nicht für jede Datenquelle ein eigener Adapter geschrieben werden muss.

Inhaltsverzeichnis

1. Einführung **1**

2. Analyse **2**
2.1. Die Service Monitoring Architecture . . . 2
2.2. Aufgabenstellung . . . 3
2.3. Anforderungen . . . 3
2.3.1. Kommunikation und Spezifikation von Schnittstellen . . . 3
2.3.2. Adaptertypen . . . 4
2.3.3. Historie . . . 4
2.3.4. Aggregation von Daten . . . 4
2.3.5. Temporale Aspekte . . . 4
2.3.6. Konfiguration über XML . . . 5
2.3.7. Asynchronität . . . 5
2.3.8. Verwalten von mehreren Datenquellen . . . 5
2.3.9. Leistung . . . 5
2.3.10. Erweiterung um Datenquellen . . . 5
2.4. Anwendungsfalldiagramm . . . 6
2.5. Zusammenfassung . . . 6

3. Entwurf **7**
3.1. Kommunikation mittels CORBA . . . 7
3.2. Schnittstellen der Adaptoren . . . 8
3.2.1. Schnittstelle der plattformunabhängigen Schicht . . . 8
3.2.2. Schnittstelle der Integrations- und Konfigurationsschicht . . . 10
3.2.3. Schnittstelle der plattformabhängigen Schicht . . . 11
3.3. Klassendiagramm der Adaptoren . . . 12
3.4. Konfiguration der Adaptoren . . . 14
3.5. Zusammenfassung . . . 15

4. Implementierung **16**
4.1. Einlesen der Konfigurationsdatei . . . 16
4.2. Initialisierung von CORBA . . . 17
4.3. Implementierung der Servants . . . 18
4.3.1. Initialisierung der Servants . . . 18
4.3.2. Allgemeines zu der Klasse AdapterSource . . . 19
4.3.3. Wichtige Anwendungsfälle . . . 20
4.4. Zugriff auf die Datenquellen . . . 25
4.5. Integration neuer Datenquellen . . . 27
4.6. Zusammenfassung . . . 28

5. Installation und Beispiel **29**
5.1. Installation des Adapters . . . 29
5.2. Beispiel . . . 31

6. Zusammenfassung **34**

A. XML Schema Definition **35**

Abbildungsverzeichnis

2.1. Die Service Monitoring Architecture [DS 05] 2
2.2. Die wichtigsten Anwendungsfälle eines Push-Adapters 6

3.1. Funktionsweise von CORBA 7
3.2. Schnittstellen zwischen der Integrations- und Konfigurationsschicht und der plattformunabhängigen Schicht 9
3.3. Abhängigkeiten der Schnittstellen 9
3.4. Schnittstelle zwischen der plattformunabhängigen Schicht und der plattformabhängigen Schicht 11
3.5. Ausschnitt aus dem Klassendiagramm für Adaptoren 13

4.1. Klassen, in denen die Konfiguration gespeichert wird 16
4.2. Abhängigkeiten der Klasse XMLParser 17
4.3. Die abstrakte Klasse AdapterSource 19
4.4. Interaktionen der start()-Methode 20
4.5. Aktivitätsdiagramm der start()-Methode in SmonaAdapter_Impl 21
4.6. Interaktionen der pullSource()-Methode 22
4.7. Die Struktur Timeout 24
4.8. Die Struktur SourceAccessData 25
4.9. Direkter Zugriff auf Datenquellen 26

Tabellenverzeichnis

1. Einführung

Die kontinuierlich ansteigende Größe von Unternehmensnetzen und die damit verbundene erhöhte Komplexität dieser Netze bezüglich deren Management hat in den letzten Jahrzehnten ein durch Software unterstütztes Management von Netz- und Systemkomponenten erfordert. Dabei spielt besonders die Überwachung von Ressourcen eine große Rolle. So können einem Unternehmen zum Beispiel Statusabfragen an einen Router ermöglichen, möglichst schnell auf Fehler, die mit diesem Router zusammenhängen, zu reagieren. Um nun die managementrelevanten Informationen von der Komplexität der konkreten Ressource zu trennen, wurde das Konzept des *Managed Object* (MO) eingeführt, welches als Abstraktion der realen Ressource zu verstehen ist. Durch das MO kann man also auf managebare Attribute einer Ressource zugreifen. Die MOs werden dabei in der sogenannten *Management Information Base* (MIB) gespeichert, anhand derer Managementwerkzeuge, um die benötigten Informationen zu erlangen, bestimmte MOs finden und abfragen können.

Damit einhergehend ist jedoch auch die Komplexität der in den Unternehmen eingesetzten Dienste sowie den von den Unternehmen den Kunden angebotenen Dienstleistungen angestiegen. Ein Dienst setzt sich allerdings aus der Funktionalität mehrere Ressourcen zusammen, demnach ist der Zustand eines Dienstes von den Zuständen einer Menge von Netz- und Systemkomponenten abhängig, ist also als Abstraktion dieser Ressourcenmenge zu verstehen. Zum Beispiel kann die Verfügbarkeit eines Dienstes von der Verfügbarkeit des Routers und der Verfügbarkeit des DNS-Servers abhängig sein. Ein auf diese Weise zusammengesetztes neu gewonnenes Attribut nennt man auch *Dienstattribut*. Durch diese Abstraktion wird das Verwalten von Diensten erleichtert, weil der Dienst als Funktionseinheit betrachtet und verwaltet wird, woraus sich Vorteile sowohl für den Anbieter, als auch für den Kunden ergeben. Somit kann zum Beispiel eine Fehlfunktion eines einem Kunden angebotenen Dienstes schneller behoben werden, was insbesondere aufgrund von *Service Level Agreements* im Sinne des Anbieters ist, da im Falle einer Nichteinhaltung der in den Vertägen abgesprochenen Konditionen entsprechende Vertragsstrafen vorgesehen sein könnten. Der Kunde profitiert von einer schnellen Lösung des Problems und der damit verbundenen Einhaltung der festgesetzten Dienstgüte.

Analog zu dem MIB Modell kann man einen Dienst in der *Service Management Information Base* (SMIB) beschreiben. Jedoch ist der herkömmliche komponentenorientierte Ansatz nicht für das Dienstmanagement geeignet, da ihm die dienstorientierte Managementsicht fehlt. So werden von bereits vorhandenen Managementwerkzeugen fast ausschließlich komponentenorientiertes Management unterstützt. Demnach wäre eine Dienstmanagementarchitektur wünschenswert, welche eine Aggregation der aus den Komponenten gewonnenen Attributen zu Dienstattributen durchführt. Dazu ist es auch notwendig, dass die benutzten Werkzeuge zur Überwachung von Ressourcen die einzelnen Attribute in einem einheitlichen Format zur Verfügung stellen.

2. Analyse

In diesem Kapitel soll auf das Ziel des Fortgeschrittenenpraktikums und auf die Anforderungen der zu erstellenden Software eingegangen werden. Zuerst wird eine Schichtenarchitektur für das Dienstmanagement vorgestellt, so dass die Einordnung der Software in das Gesamtsystem erfolgen kann. Ausgehend von der Aufgabenstellung können dann eine Reihe von Anforderungen formuliert werden. Schließlich werden noch grundlegende Anwendungsfälle vorgestellt.

2.1. Die Service Monitoring Architecture

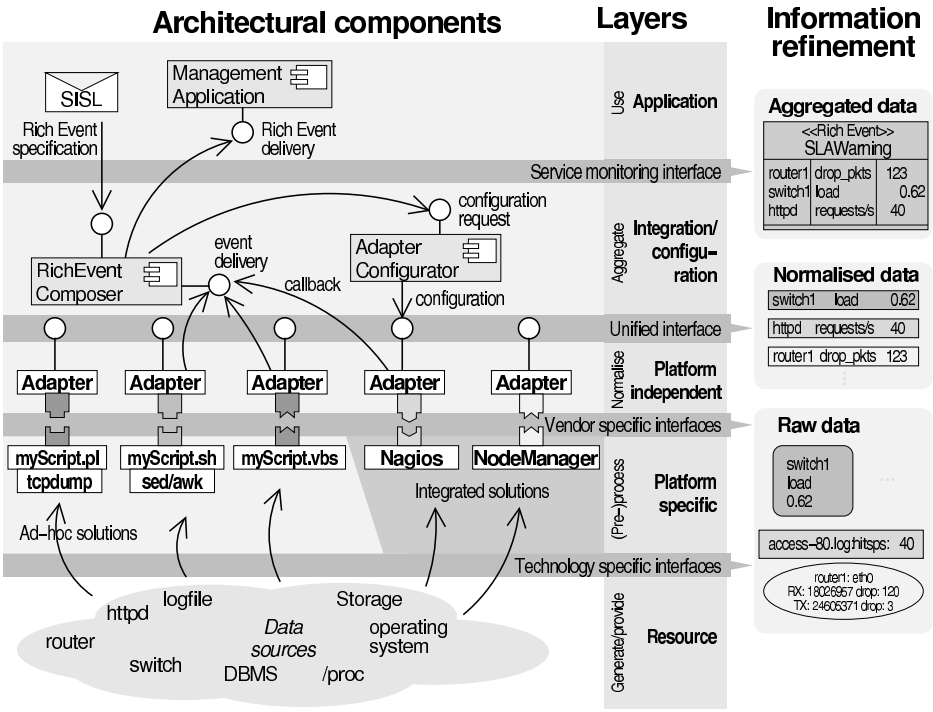

Abbildung 2.1.: Die Service Monitoring Architecture [DS 05]

In der Einführung wurde die Notwendigkeit für eine Architektur für das Dienstmanagement erläutert. Deshalb wurde am Institut für Informatik der Ludwig-Maximilians-Universität München eine Schichtenarchitektur (SMONA) entwickelt (siehe Abbildung 2.1), die auf die vorgestellten Erfordernisse eingeht. Die Aufgaben der einzelnen Schichten werden im folgenden näher beschrieben:

- *Ressourcenschicht:* Auf dieser Schicht befinden sich die zu überwachenden Datenquellen. Jede Ressource kann über mehrere Attribute verfügen. Eine Teilmenge dieser Attribute von einer Teilmenge der zur Verfügung stehenden Ressourcen bilden dann die Dienstattribute.
- *Plattformspezifische Schicht:* Hier wird der Zugriff auf die Ressourcen geregelt. Dies kann zum einen durch bereits vorhandene Software geschehen, aber auch die Benutzung eigener Software ist möglich und in bestimmte Fällen sogar notwendig, um geeignete Managementinformationen aus den Datenquellen zu extrahieren. Die dadurch erhaltenen Informationen besitzen kein einheitliches Format.

- *Plattformunabhängige Schicht:* Die auf dieser Schicht laufenden Adaptoren benutzen die plattformspezifische Schicht, um auf die Ressourcen zuzugreifen, wobei im Prinzip auch möglich ist, dass dieser Zugriff direkt im Adapter erfolgt, also die plattformspezifische Schicht umgangen wird. Die gesammelten Daten werden dann je nach Konfiguration aufbereitet und in einem einheitlichen Format über eine definierte Schnittstelle an die Integrations- und Konfigurationsschicht weitergegeben.
- *Integrations- und Konfigurationsschicht:* Während die bis jetzt vorgestellten Schichten auf demselben System laufen, kann diese Schicht auf einem anderen System implementiert sein (gewöhnlich ist dies das Management System). Voraussetzung ist eine entsprechende Vernetzung beider Systeme. Die *Adapter Configurator* Komponente ist für das Konfigurieren der Adapter verantwortlich. Hier kann zum Beispiel angegeben werden, in welchen zeitlichen Abständen welche Ressourcen abgefragt werden sollen. Der *RichEvent Composer* erhält die von dem Adapter gesendeten Attribute und führt außerdem die Aggregation dieser Attribut durch. Die daraus entstehenden Datensätze (*RichEvents*) werden an die Anwendungsschicht weitergegeben.
- *Anwendungsschicht:* Auf dieser Schicht läuft die eigentliche Managementanwendung, die durch einen Benutzer oder weiterer Software bedient werden kann. Informationen darüber, welche Attribute ein Dienstattribut bilden, welche Datenquellen dazu abgefragt werden müssen sowie Bedingungen an die Attribute werden auf dieser Schicht mit der *Service Information Specification Language* (SISL) spezifiziert und der Integrations- und Konfigurationsschicht mitgeteilt.

2.2. Aufgabenstellung

Es sollen systemnahe Adaptoren für mehrere Datenquellen auf Linux-Systemen entwickelt werden. Die Adaptoren stellen die plattformunabhängige Schicht von SMONA dar und übernehmen das Sammeln von Informationen, die von den Datenquellen angeboten werden. Die Adaptoren sind dabei für einfache Formen der Aggregation, für Formatnormalisierung sowie für den Zugriff auf die Datenquellen mit Hilfe der plattformabhängigen Schicht oder direkt über Systemaufrufe und letztendlich der Weitergabe dieser Informationen an die Integrations- und Konfigurationsschicht verantwortlich. Dadurch wird die Heterogenität der Datenquellen von der Integrations- und Konfigurationsschicht verborgen. Zudem muss eine Schnittstelle für die Kommunikation dieser Schichten spezifiziert werden. Des weiteren sollen verschiedene Adaptoren-Konzepte wie *Push*- oder *Pull*-Adaptoren sowie temporale Aspekte unterstützt werden. Es soll insbesondere auch darauf geachtet werden, dass neue Datenquellen mit möglichst wenig Aufwand zu den von den Adaptoren bereits unterstützten Datenquellen hinzugefügt werden können. Diese Konzepte werden im nächsten Kapitel erläutert.

2.3. Anforderungen

In der Aufgabenstellung wurden bereits einige Anforderungen genannt, auf die in diesem Kapitel näher eingegangen werden soll. Zusätzlich werden auch noch weitere Anforderungen vorgestellt. Neben funktionalen Anforderungen wird auch auf nicht-funktionale Anforderungen eingegangen, denen eine große Bedeutung zukommt, weil durch sie das Konzept der Adaptoren starkt beeinflusst wird. Hier ist insbesondere die Erweiterung der Adaptoren um neue Datenquellen zu nennen.

2.3.1. Kommunikation und Spezifikation von Schnittstellen

Es besteht eine Notwendigkeit zur Kommunikation zwischen der plattformunabhängigen Schicht und der Integrations- und Konfigurationsschicht. Zum einen müssen Konfigurationsanfragen oder auch Statusabfragen an die Adaptoren gesendet werden können, zum anderen müssen auch die von den Adaptoren gelesenen Werte zur weiteren Bearbeitung übertragen werden. Demnach ist eine Spezifikation von Schnittstellen für die jeweilige Schicht erforderlich.

Es soll angemerkt werden, dass sich die Schichten gewöhnlich auf unterschiedlichen Rechnern in Ausführung befinden. Die Adaptoren laufen auf den Komponenten, die überwacht werden sollen, die Integrations- und

Konfigurationsschicht auf der anderen Seite kann zum Beispiel auf einem Rechner implementiert sein, der als Managementkonsole dient und die unmittelbar mit der Managementanwendung auf Anwendungsebene kommuniziert.

Die plattformunabhängige Schicht nimmt also sowohl die Rolle eines Servers, als auch eines Clients an. Das Selbe gilt für die Integrations- und Konfigurationsschicht.

2.3.2. Adaptertypen

Es sollen zwei Typen von Adaptoren unterstützt werden. Nehmen wir an, eine Datenquelle soll überwacht werden und der Adapter genau dann ein Ereignis auslösen, wenn ein festgesetzter Schwellenwert überschritten wird. Die Initiative für die Kommunikation geht hierbei vom Adapter aus. Solche Adapter nennen wir Push-Adapter. Ein Beispiel für diesen Adaptertyp ist die Überwachung der Netzauslastung. Wenn die Auslastung über einen bestimmten Schwellenwert steigt, kann dem Kunden aufgrund der höheren Verzögerung unter Umständen nicht mehr die in den Verträgen festgesetzte Mindestgeschwindigkeit garantiert werden. Auf solche Fälle muss entsprechend reagiert werden.

Ein anderer Adaptertyp ist der Pull-Adapter, bei dem die Initiative für die Kommunikation von der Konfigurations- und Integrationsschicht ausgeht. Hierbei wird explizit bei den Adaptoren ein Wert abgefragt.

Der dritte Typ ist der Set-Adapter. Dieser dient nicht der Überwachung von Datenquellen, sondern zum verändern managementrelevanter Parameter in den Komponenten. Dieser Adaptertyp wird noch nicht unterstützt.

2.3.3. Historie

Bei dem eben angesprochenen Pull-Adapter erfragt die Konifgurations- und Integrationsschicht nach dem aktuellen Wert eines Attributs einer Datenquelle. Der Pull-Adapter muss daraufhin die Datenquelle abfragen. Dieser Vorgang kann jedoch mehr Zeit in Anspruch nehmen, als die anfragende Schicht bereit ist zu warten. So könnte zum Beispiel vorkommen, dass im Moment noch keine Werte bei der Datenquelle vorliegen, die anfragende Schicht jedoch möglichst schnell ein Ergebnis vorliegen haben muss. Um eine lange Wartezeit zu umgehen, sollte es daher möglich sein, dass ein Pull-Adapter Historien verwaltet. Wenn ein Pull-Adapter entsprechend konfiguriert wird, wird daher die Datenquelle in regelmäßigen Intervallen abgefragt und die erhaltenen Werte zwischengespeichert. Wenn der Adapter nun eine explizite Pull-Anfrage erhält, wird einfach der Wert aus diesem Zwischenspeicher geholt und sofort gesendet.

2.3.4. Aggregation von Daten

Eine weitere funktionale Anforderung an Adapter ist die Aggregation von Daten, was aber getrennt von der Aggregationsaufgabe der Integrations- und Konfigurationsschicht gesehen werden muss, die hier noch mehr Möglichkeiten bietet. Auf Adapterebene sollen Operationen auf einer Datenmenge ausgeführt werden können. Wenn zum Beispiel die gemessenen Werte einer Datenquelle großen Schwankungen ausgesetzt sind, ist es nützlich, aus mehreren Meßwerten den Mittelwert zu bilden und diesen an die nächsthöhere Schicht zu liefern. Dadurch wird das Datenaufkommen zwischen den beiden Schichten gering gehalten, was sich postitiv auf die Netzlast auswirkt. Diese Mittelwertbildung sowie weitere Operationen müssen Adapter unterstützen.

2.3.5. Temporale Aspekte

Hierbei werden zwei Aufgaben unterschieden: Zum einen muss für einen Push-Adapter oder einen Pull-Adapter mit Historie angegeben werden, in welchem Intervall die Datenquelle abgefragt werden soll. Zum anderen sollen für Pull-Anfragen auch Timeouts angegeben werden können. Dies ist offensichtlich nur für Pull-Adapter ohne Historien sinnvoll und dient der Integrations- und Konfigurationsschicht dazu, bestimmte

Zeitbeschränkungen einzuhalten. Wenn nach Ablauf des Timeouts noch keine Daten der Datenquelle vorliegen, wird die Messung abgebrochen und eine Fehlermeldung an die Integrations- und Konfigurationsschicht zurückgegeben.

2.3.6. Konfiguration über XML

In Kapitel 2.1 wurde schon angemerkt, dass der Adapter Configurator für die Konfiguration der Adapter verantwortlich ist. Ein Beispiel für einen Konfigurationsparameter ist der zeitliche Abstand, in dem die Datenquelle ausgelesen werden soll. Es wäre aber nicht sinnvoll, wenn der Adapter hier beliebige Werte zulassen würde. Insofern soll der Adapter bestimmen können, welche Werte dafür überhaupt erlaubt sind. Diese Parameter soll der Adapter aus einer Konfigurationsdatei einlesen. Zusätzlich gibt es noch viele weitere Eigenschaften, die man spezifizieren muss, wie zum Beispiel den Adaptertyp. Eine Konfigurationsdatei im XML Format ermöglicht dabei eine hohe Flexibilität.

2.3.7. Asynchronität

Der Aufrufer einer Methode eines entfernten Objekts muss warten, bis entweder das Ende der Methode erreicht ist oder diese explizit verlassen wird. Wenn nun die Methode Funktionen aufruft, die blockieren können, blockiert dementsprechend auch der Aufrufer. Ein Beispiel dafür ist eine Pull-Anfrage ohne Historie, die, wie eben besprochen, einige Zeit in Anspruch nehmen kann. Um eine Blockierung des Aufrufers zu umgehen, werden daher solche Aufgaben von extra dafür vorgesehenen Threads erledigt.

2.3.8. Verwalten von mehreren Datenquellen

Ein Adapter soll den Zugriff auf mehr als eine Datenquelle unterstützten. Dadurch müssen weniger Adapterinstanzen gestartet werden, was zu einer besseren Speicherauslastung des Systems führt. Eine von einem Adapter verwaltete Datenquelle nennen wir auch Adapter-Datenquelle. Die Unterscheidung zwischen Adapter-Datenquelle und Adapter ist wichtig, da ein Adapter sozusagen als Container dient und deshalb die Adapter-Datenquellen separat konfiguriert werden müssen.

2.3.9. Leistung

Aufgrund der möglichen Einsatzgebiete der Adaptoren in Umgebungen, in denen Leistung eine große Rolle spielt, soll auch auf die Effizienz der Adaptoren geachtet werden. Daher muss es möglich sein, durch Umgehung der plattformabhängigen Schicht direkt auf die Datenquellen zuzugreifen. Dies erfordert jedoch Zugriffe auf die Schnittstelle des Betriebssystems, darunter auch Funktionen, die von Programmiersprachen wie z.B. Java nicht unterstüzt werden. Deshalb bietet sich als Sprache für die Implementierung der Adaptoren C++ an, da zudem auch Übersetzer für C++ existieren, die effizienten Code erzeugen.

2.3.10. Erweiterung um Datenquellen

Es gibt eine Vielzahl von Datenquellen, die sich sowohl im Zugriffsverfahren, als auch im Typ der zurückgelieferten Daten unterscheiden. Ein Adapter sollte die Möglichkeit bieten, mit geringem Aufwand neue Datenquellen zu integrieren. Hierbei muss zwischen Datenquellen, die indirekt über die plattformabhängige Schicht oder aber direkt über Systemaufrufe ausgelesen werden, unterschieden werden. In beiden Fällen spielt besonders die Unterstützung unterschiedlicher Datentypen eine Rolle, wobei dies durch Eigenschaften der Sprache C++ vereinfacht wird, wie z.b. die Nutzung von Templates. Wenn die Unterstützung für den Datentyp einer Datenquelle schon implementiert ist, muss für eine Integrations der Datenquelle nur im Fall eines direkten Zugriff auf die Datenquelle eine Änderung an dem Adapter durchgeführt werden, was jedoch durch die objektorientierten Spracheigenschaften von C++ vereinfacht wird.

2.4. Anwendungsfalldiagramm

Abbildung 2.2 zeigt die wichtigsten Interaktionen der Aktoren, die Informationen mit einem Push-Adapter austauschen. Die Aktoren sind dabei die einzelnen Komponenten der Integrations- und Konfigurationsschicht. Die Anwendungsfälle sind die zentralen Anforderungen an die Schnittstelle zwischen diesen beiden Schichten und es wird in den nächsten Kapiteln detailliert auf diese eingegangen.

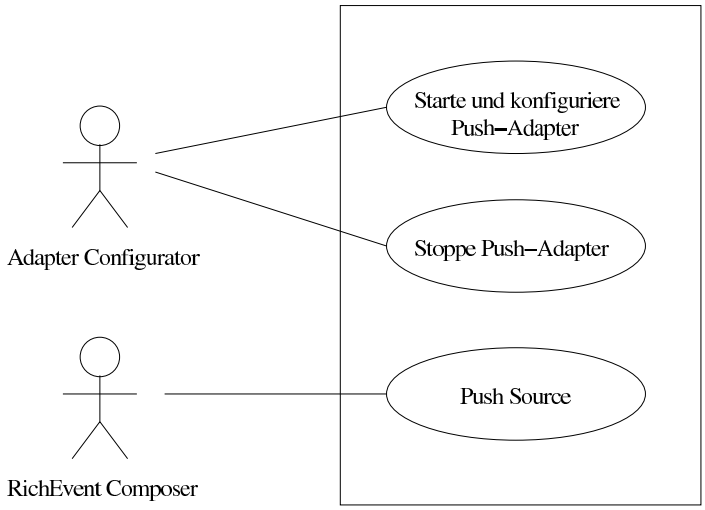

Abbildung 2.2.: Die wichtigsten Anwendungsfälle eines Push-Adapters

2.5. Zusammenfassung

In diesem Kapitel wurde die Dienstmanagementarchitektur SMONA vorgestellt. Das Ziel des Fortgeschrittenenpraktikums ist die Implementierung der plattformunabhängigen Schicht dieser Architektur, deren Aufgabe es ist, die Heterogenität der Datenquellen zu verbergen und eine einheitliche Schnittstelle den höheren Schichten zur Verfügung zu stellen. Nachdem in der Aufgabenstellung schon Anforderungen an die Software genannt wurden, wurde diese Liste schließlich erweitert und die Anforderungen genauer spezifiziert. Danach wurden die wichtigsten Anwendungsfälle eines Push-Adapters genannt, die erheblich zum Verständnis der Funktionsweise der Adaptoren beitragen. Sie werden ausführlich in Kapitel 4 beschrieben.

3. Entwurf

In der Entwurfsphase soll auf die Schnittstellen näher eingegangen werden (siehe Kapitel 2.3.1). Die angebotenen Dienste ergeben sich aus den in Kapitel 2.4 vorgestellten Anwendungsfällen. Zusätzlich werden noch einige triviale Funktionen angeboten. Um die strukturellen Eigenschaften der Adaptoren zu verdeutlichen, wird danach ein Ausschnitt aus dem Klassendiagramm der Adaptoren vorgestellt. Zusätzlich sollen noch die Konfigurationsmöglichkeiten der Adaptoren beschrieben werden. Eine genaue Beschreibung der Anwendungsfälle findet dann erst in Kapitel 4 im Zusammenhang der Implementierung statt.

3.1. Kommunikation mittels CORBA

In der Beschreibung der Anforderungen wurde bereits auf die Notwendigkeit zur Netzkommunikation zwischen den Schichten hingewiesen. Um keine eigenen Anwendungsprotokolle entwickeln zu müssen, was einen hohen zusätzlichen Arbeitsaufwand zur Folge hätte, und um nicht auf die vom Betriebssystem bereitgestellten Funktionen zur Netzkommunikation zugreifen zu müssen, was aufgrund der Heterogenität der Systeme zu Kompatibilitätsproblemen führen würde, wird zur Kommunikation eine Middlewareplattform zur Untersützung verteilter Anwendungen verwendet. Die Wahl ist hierbei auf die *Common Object Request Broker Architecture* (CORBA) gefallen, eine frei verfügbare, objektorientierte Middleware. CORBA ist relativ leistungsstark und bietet außerdem die benötigte Funktionalität an. Da in den nächsten Kapiteln einige CORBA-spezifische Begriffe verwendet werden, soll zuerst eine kurze Einführung in die Funktionsweise von CORBA und des von CORBA angebotenen *Naming Service* erfolgen.

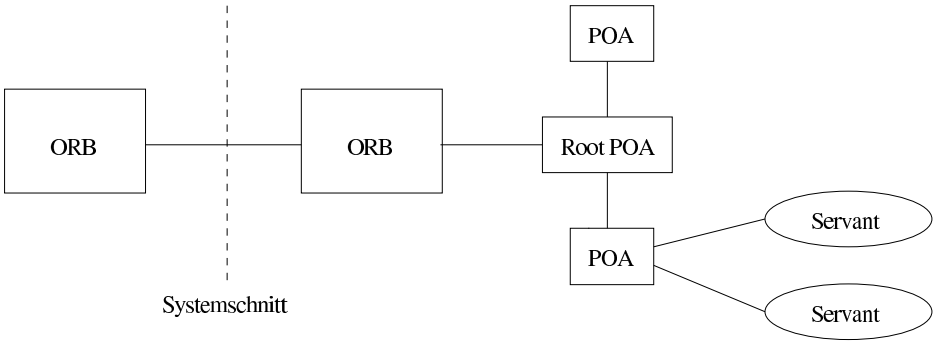

Abbildung 3.1.: Funktionsweise von CORBA

Grundlegendes Konzept von CORBA ist das *CORBA Objekt*, welches eine Schnittstelle für Dienste bereitstellt, die von einem Client benutzt werden können. Der Client stellt dabei eine Anfrage, die an den Server, auf dem sich das Objekt befindet, gesendet wird. Nachdem die entsprechende Methode ausgeführt wurde, wird das Ergebnis an den Client zurückgesendet. Dieser Vorgang geschieht völlig transparent. Die Transparenz wird durch den *Object Request Broker* (ORB) erreicht. Der ORB auf der Serverseite empfängt dabei die Anfragen von dem ORB auf Clienseite. Zusätzlich kommuniziert der ORB auf Serverseite mit dem *Portable Object Manager* (POA), der die CORBA Objekte verwaltet. CORBA Objekte besitzen eine Vielzahl von Eigenschaften, wobei diese Eigenschaften an den POA gebunden sind. Wenn also eine Anwendung CORBA Objekte mit verschiedenen Eigenschaften unterstützen soll, müssen dafür mehrere POAs mit den gewünschten Eigenschaften erzeugt werden. Diese werden von dem Root-POA verwaltet, der selbst auch CORBA Objekte verwalten kann, dann aber nur mit den Standard-Eigenschaften. Dieses Konzept wird in Abbildung 3.1 veranschaulicht.

Die Schnittstelle eines CORBA Objekts wird mit der *Interface Definition Language* (IDL) beschrieben und sind unabhängig von einer Programmiersprache. Ein IDL-Compiler übersetzt dann die Datei mit der Schnittstellenbeschreibung in eine konkrete Programmiersprache. Die daraus entstehende Datei wird Stub (für den Client) oder Skeleton (für den Server) genannt. Es gilt zu beachten, dass ein CORBA Objekt eine abstrakte Einheit ist, das konkrete Objekt existiert also erst, nachdem die vom Skeleton angebotene Schnittstelle implementiert wurde. Solch ein Objekt wird *Servant* genannt. Servants sind also im Prinzip nichts anderes, als die Objekte einer Klasse, welche die Schnittstelle des CORBA Objekts implementieren. Die IDL-Datei wird vom Server für die Implementierung des Servants benötigt, und vom Client, damit dieser weiß, welche Dienste das entsprechende Objekt unterstützt.

Nun stellt sich die Frage, auf welche Weise der Client von der Existenz der CORBA Objekte erfährt und wie er auf deren Dienste zugreifen kann. Ein CORBA Objekt wird mit einer *Objektreferenz* identifiziert. Diese enthält nicht nur das entsprechende Objekt, sondern auch die benötigten Verbindungsinformationen wie IP-Adresse des Rechners, auf dem sich das Objekt befindet und die Portnummer, auf der der ORB auf Verbindungen wartet. Wenn man also die Objektreferenz kennt, muss man nichts weiteres tun, als eine bestimmte Methode des entsprechenden Objekts aufzurufen. Alle andere Arbeit wird von CORBA übernommen. An dieser Stelle soll erwähnt werden, dass diese Methodenaufrufe synchron sind, der Client also blockt, bis diese beendet wurden und der Client-ORB vom Server-ORB das Resultat empfangen hat. An dieser Stelle soll darauf hingewiesen werden, dass CORBA auch asynchrone Aufrufe über das *Dynamic Invocation Interface* anbietet, was jedoch von den Adaptoren nicht unterstützt wird. Die Publizierung der Objektreferenzen geschieht auf Serverseite und wird von dem CORBA Naming Service übernommen. Objektreferenzen werden dabei analog zu Dateien gehandhabt, der Naming Service besitzt also eine Baumstruktur. Die Knoten spielen die Rolle der Verzeichnisse, die Blätter der Dateien, also der Objektreferenzen, die damit durch einen Pfadnamen adressierbar sind. Dementsprechend verfügt jeder Adapter über eine eindeutige Kennung von allen seinen unterstützten Datenquellen. Diese Kennung ist der Pfadname, der im Naming Service zusammen mit der dazugehörigen Objektreferenz gespeichert wird und wird im Verlauf des Dokuments als *Source ID* bezeichnet. Es sei darauf hingewiesen, dass mehrere Pfadnamen auf dieselbe Objektreferenz verweisen können. Der Naming Service wird von eine Programm implementiert, welches man *CORBA Nameserver* nennt. Der Client erhält nun die Objektreferenzen, indem er IP-Adresse und Portnummer eines Rechners kennt, auf dem ein CORBA Nameserver läuft und eine Anfrage mit der Kennung der Adapter-Datenquelle stellt. Für weitere Informationen über CORBA wird auf [HV 99] verwiesen.

3.2. Schnittstellen der Adaptoren

Die plattformunabhängige Schicht nutzt zum einen die Dienste der plattformabhängigen Schicht, um selbst die Plattformunabhängigkeit zu erhalten, zum anderen bietet sie der Integrations- und Konfigurationsschicht entsprechende Dienste an und nutzt wiederum Dienste der Integrations- und Konfigurationsschicht, um die Ergebnisse an diese liefern zu können. Im nachfolgenden wird auf diese drei Schnittstellen eingegangen.

3.2.1. Schnittstelle der plattformunabhängigen Schicht

Die Zusammenhänge werden in Abbildung 3.2 genauer vorgestellt. Um eine Kommunikation unter Verwendung von CORBA zwischen diesen Schichten zu ermöglichen, ist eine Beschreibung dieser Schnittstellen mit der IDL notwendig. Da es jedoch mehrere Adaptertypen gibt, können die Schnittstellen voneinander abweichen. Diese Abhängigkeiten werden in Abbildung 3.3 veranschaulicht. Die Funktionalität von *SmonaAdapter* wird von allen Adaptoren genutzt, zusätzlich gibt es noch für Push- und Pulladaptoren eigene Schnittstellen, welche die Funktionalität von *SmonaAdapter* erben. *PushAdapter* und *PullAdapter* werden dann von den Adaptoren implementiert. Nachfolgend sollen die Methoden dieser Schnittstellen vorgestellt werden.

1. Schnittstelle für *SmonaAdapter*:
   - *void start(in string source_id, in BasicConfiguration basic_conf)*
     Startet und konfiguriert eine vom Adapter unterstützte Datenquelle. Die Datenquelle wird mit *source_id* identifiziert. Es wird empfohlen, dass *source_id* mit dem Pfad, unter dem die Datenquelle im Naming Service publiziert wurde, übereinstimmt. *basic_conf* enthält die Konfigurationsparameter,

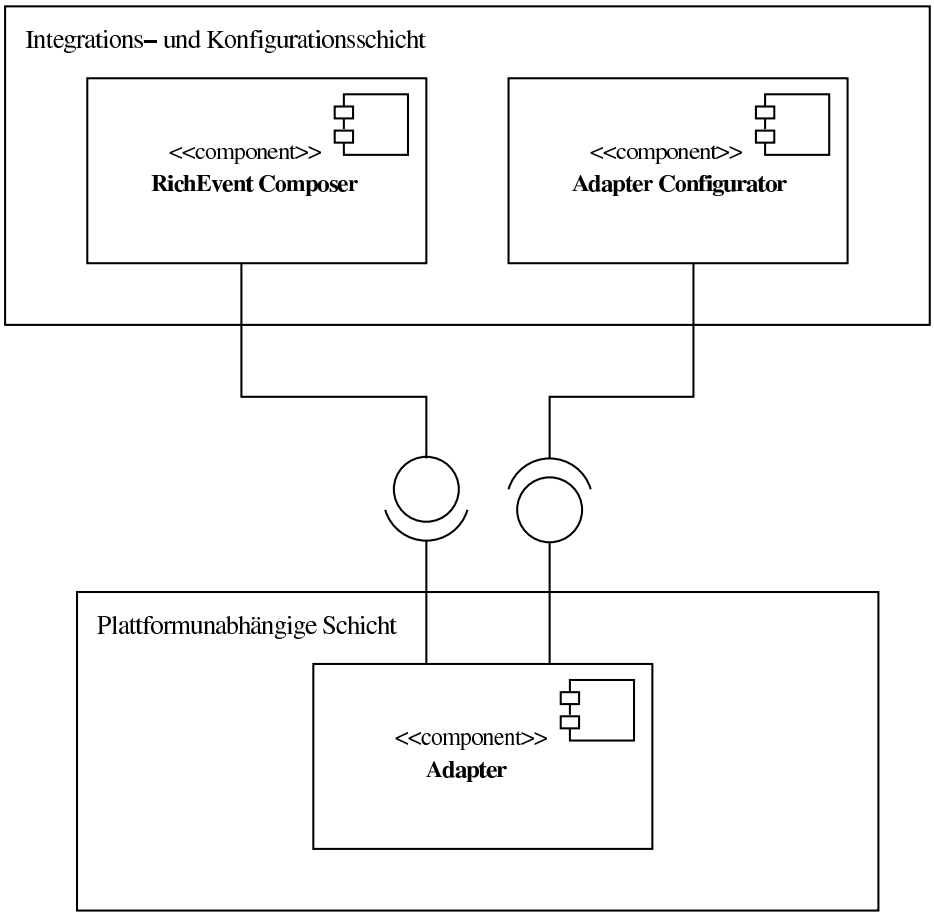

Abbildung 3.2.: Schnittstellen zwischen der Integrations- und Konfigurationsschicht und der plattformunabhängigen Schicht

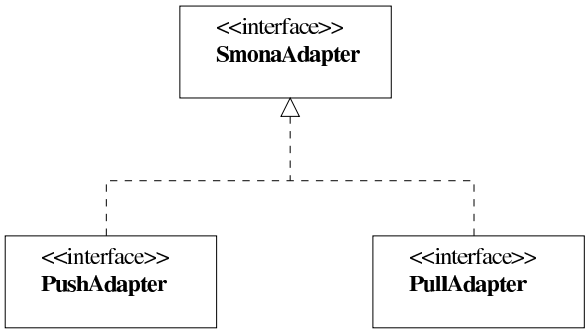

Abbildung 3.3.: Abhängigkeiten der Schnittstellen

wie zum Beispiel das Intervall, gemäß dem auf die Datenquelle zugegriffen wird. Der Datentyp *BasicConfiguration* ist eine Liste des Datentyps *Property*, welches ein *name/value* Paar vom Typ *string/CORBA::Any* darstellt, wobei *CORBA::Any* eine spezielle CORBA-Klasse ist, deren Objekte Daten beliebigen Typs speichern können. In dem Intervall-Beispiel enthält dann *name* einen Bezeichner für Intervall und *value* den eigentlichen Wert. Die Bezeichner für *name* sind beliebig und es können grundsätzlich beliebige Eigenschaften anhand von *BasicConfiguration* übermittelt werden. Es können mehrere Fehler beim Ausführen der *start()*-Methode auftreten. In diesem Fall werden entsprechende Exceptions geworfen, die vom Client abgefangen werden können.

- *void stop(in string source_id)*
  Stoppt eine schon konfigurierte Adapter-Datenquelle. Ein Fehler tritt auf, wenn *source_id* ungültig ist. Wenn *start()* noch nicht aufgerufen wurde, passiert nichts.

- *any getSelfDescription(in string source_id)*
  Es wäre wünschenswert, wenn die Integrations- und Konfigurationsschicht Eigenschaften der Adapter-Datenquellen, welche durch *source_id* gegeben ist, abfragen könnte, also zum Beispiel, in welchem Bereich das Intervall liegen muss. Daher sollte ein Adapter selbst seine Eigenschaften beschreiben und öffentlich machen können, was durch diese Methode realisiert wird. Allerdings ist sie noch nicht implementiert, bzw. liefert nur einen Dummy-Wert zurück.

- *string getVersion()*
  Liefert die Adapter-Version zurück.

- *string getId()*
  Liefert die Adapter-Kennung zurück.

- *any getProperty(in string name, in string source_id)*
  Liefert den Wert einer mit *name* bezeichneten Property der entsprechenden Adapter-Datenquelle zurück. Für die Integrations- und Konfigurationssicht gibt es für dies Funktion noch keinen Anwendungsfall, daher ist diese Methode noch nicht implementiert, bzw. liefert nur einen Dummy-Wert zurück.

2. Schnittstelle für *PushAdapter*:
   Ein Push-Adapter bietet zwar zusätzliche Funktionalität an, wie zum Beispiel die Unterstützung von Schwellwerten, jedoch erfordert dies keine Erweiterung der Schnittstelle um Methoden, da dies einfach durch einen weiteren Konfigurationsparameter gelöst werden kann.

3. Schnittstelle für *PullAdapter*:

   - *void pullSource(in string source_id)*
     Frage explizit eine Datenquelle ab. Der gelesene Wert wird zurückgeliefert. Kann nur ausgeführt werden, wenn zuvor die mit *source_id* bezeichnete Adapter-Datenquelle gestartet wurde.

3.2.2. Schnittstelle der Integrations- und Konfigurationsschicht

Zusätzlich zu den von den Adaptoren angebotenen Funktionen, muss es auch auf der Seite der Integrations- und Konfigurationsschicht eine Schnittstelle geben, so dass die Adaptoren die gelesenen Werte an diese Schicht weitergeben können. Dies ist auch in Abbildung 3.2 veranschaulicht. Die dafür mit der IDL definierten Schnittstelle heißt *SmonaListener*, welche jedoch leer ist. *PushListener* und *PullListener* erben von *SmonaListener*. Die entsprechende Objektreferenz wird dem Adapter mit der Methode *start()* über einen zusätzlichen Eintrag in *BasicConfiguration* mitgeteilt. Die unterstützten Methoden werden nun kurz vorgestellt.

1. Schnittstelle für *PushListener*:

   - *void pushEvent(in string source_id, in any value)*
     Wird aufgerufen, um einen von einer Datenquelle gelesenen Wert weiterzugeben. Dieser Wert ist in dem CORBA::Any Objekt *value* gespeichert. Die Integrations- und Konfigurationsschicht muss diesen Wert aus dem Objekt extrahieren. Der Wert für *source_id* ist der in der *start()* Methode

angegebene Wert. Dies ist nützlich, da somit die Integrations- und Konfigurationsschicht mehreren Adapter-Datenquellen dieselbe Objektreferenz mitteilen kann. Von welcher Datenquelle das Ergebnis kommt, kann man anhand von *source_id* bestimmen.

- *void send_SourceError(in string source_id, in string src_err, in long code)*
  Ermöglicht das senden von Fehlern, die während der Überwachung von Datenquellen eintreten können. *src_err* ist ein String, der den Fehler beschreibt und optional ist. *code* ist der Exit-Code der den Fehler verursachenden Operation. Wie die Integrations- und Konfigurationsschicht auf diese Fehler reagiert, ist nicht spezifiziert.

2. Schnittstelle für *PullListener*:

   - *void send_Value(in string source_id, in any value)*
     Analog zu *pushEvent*.
   - *void send_SourceError(in string source_id, in string src_err, in long code)*
     Analog zu der Methode in *PushListener*.

3.2.3. Schnittstelle der plattformabhängigen Schicht

Nachdem die Dienste der Schicht oberhalb der plattformunabhängigen Schicht ausführlich vorgestellt wurden, ist es jetzt an der Zeit, genauer auf die Schicht unterhalb der plattformunabhängigen Schicht einzugehen. Das Konzept wird in Abbildung 3.4 vorgestellt. Die Abbildung zeigt, dass der Zugriff auf eine Datenquelle

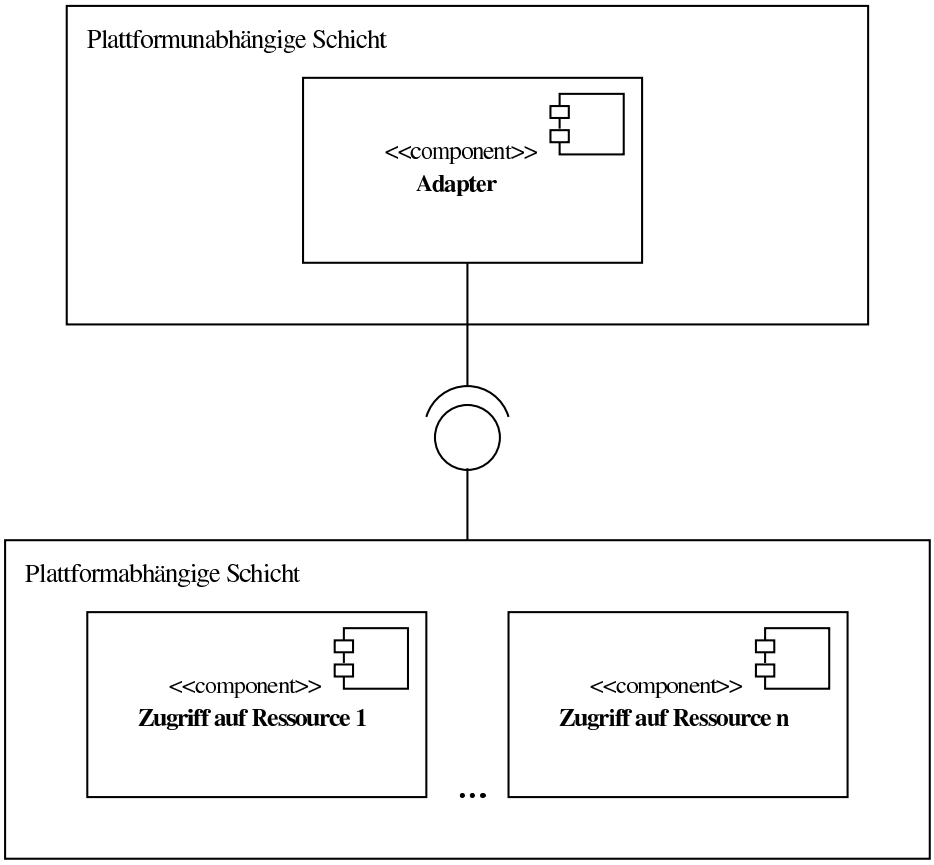

Abbildung 3.4.: Schnittstelle zwischen der plattformunabhängigen Schicht und der plattformabhängigen Schicht

nicht zwingend im Adapter, sondern auch in der plattformabhängigen Schicht stattfinden kann, woraus sich mehrere Vorteile ergeben. So gibt es zum einen mehrere Möglichkeiten, auf Datenquellen zuzugreifen, wie

zum Beispiel die Verwendung von schon bestehenden Programmen oder durch auslesen des speziellen Unix-Dateisystems *proc*. Solche Zugriffsmöglichkeiten können sich aber ändern. Das Ausgliedern des konkreten Dantenquellenzugriffs ermöglicht somit eine Änderung während des laufenden Betriebs eines Adapters. Unmittelbar damit hängt das Einfügen von neuen Datenquellen zusammen. Die Unterstützung von neuen Datenquellen erfordert zwar auch eine Änderung der Konfiguration des Adapters, allerdings ist keine Änderung am Quelltext notwendig. Davon gibt es allerdings auch Ausnahmen, zum Beispiel wenn das Format der von der neuen Ressource gelieferten Daten vom Adapter noch nicht unterstützt wird. Der Zugriff auf eine Datenquelle findet also durch einen externen Aufruf einer Komponente der plattformabhängigen Schicht statt. Der Adapter kann dann den von der Komponente gelesenen Wert abfragen, der aber noch in den korrekten Typ konvertiert werden muss, der dann schließlich auch von der Integrations- und Konfigurationsschicht erwartet wird. Allerdings bietet dieses Vorgehen auch einen großen Nachteil, denn externe Programmaufrufe sind sehr teuer. Daher ist es notwendig, den Datenquellenzugriff auch innerhalb der Adaptoren zu erlauben. Um in diesem Fall eine große Menge an unterschiedlichen Datenquellen ansprechen zu können, sollten die Adaptoren Zugang zu möglichst vielen Funktionen des zugrundeliegenden Betriebssystems haben. Daher bietet sich als Implementierungssprache besonders C++ an.

3.3. Klassendiagramm der Adaptoren

Als nächstes soll ein Ausschnitt aus dem Klassendiagramm (siehe Abbildung 3.5) für die Adaptoren vorgestellt und erklärt werden, denn die Zusammenhänge und die Beschreibung der Aufgaben der im Adapter verwendeten Komponenten führt zu einem besseren Verständnis der Funktionsweise der Adaptoren.

SmonaAdapter, *PushAdapter* and *PullAdapter* sind die aus den in Kapitel 3.2.1 vorgestellten IDL-Dateien erzeugten Skeletons. Alle der in den Skeletons deklarierten Methoden sind abstrakt und müssen daher erst implementiert werden. Diese Implementierung erfolgt in den Klassen *SmonaAdapter_Impl*, *PushAdapter_Impl* und *PullAdapter_Impl*. Es gilt zu beachten, dass sowohl *PushAdapter_Impl* als auch *PullAdapter_Impl* alle Methoden aus *SmonaAdapter* implementiert, da *PushAdapter* und *PullAdapter* diese Methoden erben. Diese beiden Klassen stellen also die Servants dar. Um jedoch Programmcode von Push- und Pull-Adaptoren gemeinsam nutzen zu können, werden die Methoden aus *SmonaAdapter* in *SmonaAdapter_Impl* implementiert und an *PushAdapter_Impl* und *PullAdapter_Impl* weitervererbt. In *PushAdapter_Impl* und *PullAdapter_Impl* werden daher diese Methoden nur als Wrappermethoden benutzt, welche dann die geerbten Methoden aus *SmonaAdapter_Impl*, in denen die eigentliche Arbeit passiert, aufrufen.

Es wurde schon in Kapitel 2.3.10 darauf hingewiesen, dass die von den Datenquellen gelieferten Informationen ein unterschiedliches Format haben können. Aufgabe der Adaptoren ist, ein einheitliches Format an der Schnittstelle zu der Integrations- und Konfigurationsschicht anzubieten. Dazu werden die gelesenen Informationen in einen geeigneten Typ umgwandelt und in ein *CORBA::Any*-Objekt geschrieben. Diese Kapselung ermöglicht eine einfache Kommunikation zwischen Client und Server. Es gibt jedoch auch zahlreiche Funktionen, die auf konkreten Typen operieren müssen. Ein Beispiel wäre die in Abschnitt 2.3.4 erwähnte Ermittlung des Mittelwerts einer Datenmenge. Dementsprechend wäre es wünschenswert, wenn möglichst viele Typen von den Adaptoren unterstützt werden würden und eine Erweiterung um weitere Typen nur geringe Änderungen im Programmcode mit sich bringen würde. Dies wird durch die Template-Klasse *PushAdapterSource* und die im Diagramm nicht gezeigte Template-Klasse *PullAdapterSource* erreicht. Diese Klassen können mit beliebigen Typen instanziiert werden. Im Klassendiagramm wird dies durch die Liste *PushAdapterSource<Integer>*, *PushAdapterSource<Double>* usw. dargestellt.

Diese instanziierten Klassen sind im Fall von Push-Adaptoren für die Erzeugung von Monitor-Objekten sowie für die Erzeugung von Push-Objekten verantwortlich. Erstere sind für die Überwachung der Datenquelle zuständig, greifen also in regelmäßigen Abständen auf die Datenquelle zu. Für diesen Zugriff werden Objekte der Klasse *SourceAccess* benutzt. *SourceAccess* kann die Datenquellen entweder indirekt über die plattformabhängige Schicht auslesen, oder aber direkt durch die Möglichkeit, auf sämtliche Funtionen des Betriebssystems zugreifen zu können. Die Push-Objekte erhalten die vom Monitor-Objekt gemessenen Daten und leiten sie an die Integrations- und Konfigurationsschicht durch Verwendung eines Push-Listeners weiter. Pull-Adaptoren funktionieren im Prinzip genauso, mit den Unterschieden, dass ein Monitor-Objekt nur für einen Pull-Adapter mit Historie sinnvoll ist und ein entsprechendes Pull-Objekt nur eine einzige Anfrage ausführt

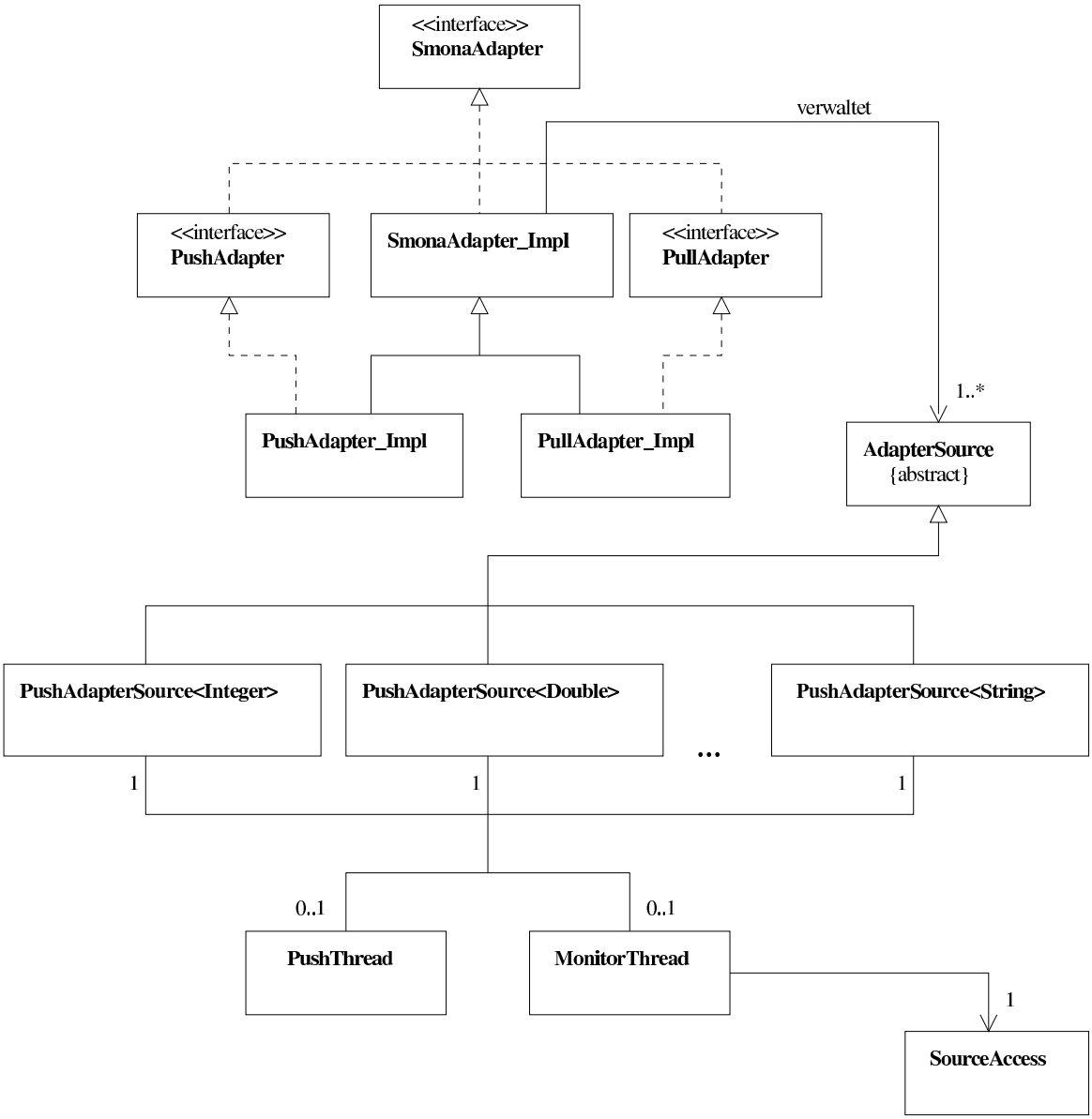

Abbildung 3.5.: Ausschnitt aus dem Klassendiagramm für Adaptoren

und danach wieder zerstört wird.

SmonaAdapter_Impl selbst steht in Beziehung zu einer oder mehreren Instanzen der abstrakten Klasse *AdapterSource*. Von dieser erben *PushAdapterSource* und *PullAdapterSource*. Dadurch wird ermöglicht, dass *SmonaAdapter_Impl* nur mit einer Klasse arbeiten muss. Wenn auf einem Objekt dieser Klasse Methoden aufgerufen werden, wird durch dynamische Bindung die korrekte Methode der instanziierten Template-Klasse aufgerufen. Details dazu in Kapitel 4.3.2.

3.4. Konfiguration der Adaptoren

In Kapitel 2.3.6 wurde auf die Vorteile einer Verwendung von Konfigurationsdateien im XML Format hingewiesen. In diesem Kapitel wird nun das Layout der Konfigurationsdateien sowie sämtliche verwendeten Elemente und deren Bedeutung beschrieben (nach [Duerr 06]).

- `<adapter>...</adapter>`
  Jedes XML Dokument benötigt ein Root-Element, wobei alle anderen Elemente innerhalb dieses Elements stehen. `<adapter>...</adapter>` ist das Root-Element der Konfigurationsdatei der Adaptoren. Es unterstützt ein Attribut dessen Angabe notwendig ist und die ID des Adapters enthält. Wie diese IDs auszusehen haben, ist noch nicht spezifiziert. Der Wert der ID kann vom Client mit Hilfe der Methode *getId()*, die jeder Adapter implementieren muss, abgefragt werden.

- `<push_adapter>...</push_adapter>` und `<pull_adapter>...</pull_adapter>`
  Innerhalb des Root-Elements ist genau ein Element zulässig. Dieses idenitifiziert den Adaptertyp und hat entweder den Namen `<push_adapter>...</push_adapter>` für einen Push-Adapter oder `<pull_adapter>...</pull_adapter>` für einen Pull-Adapter. Prinzipiell ist auch die Angabe von `<set_adapter>...</set_adapter>` erlaubt, allerdings bricht der Adapter in diesem Fall mit einer Fehlermeldung ab, da die Unterstützung für Set-Adaptoren noch nicht implementiert ist.

- `<source>...</source>`
  Dies ist das einzige Element, welches innerhalb von `<push_adapter>...</push_adapter>` oder `<pull_adapter>...</pull_adapter>` stehen kann, und beschreibt eine Adapter-Datenquelle. Es können jedoch beliebig viele `<source>...</source>` Elemente angegeben werden, es werden also im Prinzip beliebig viele Adapter-Datenquellen unterstützt. Das Attribut *id* gibt den String an, der diese Datenquelle identifiziert. Dieser String muss angegeben werden und wird zum Beispiel vom Naming Service verwendet. Außerdem kann `<source>...</source>` einfache und komplexe Typen beinhalten. Die einfachen Typen sind zum einen `<src_type>...</src_type>`, was den Typ der von den Ressourcen gelieferten Daten angibt. Momentan werden nur Standard-Typen unterstützt, nämlich *short*, *int*, *long*, *float*, *double*, *string* und *boolean*. Zum anderen gibt es das Element `<description>...</description>`, dessen Daten eine Beschreibung des Adapters darstellen.

- `<interval>...</interval>`
  Das Element `<interval>...</interval>` ist eines der komplexen Typen und enthält Informationen darüber, wie die Intervall-Werte des Clients interpretiert werden sollen. Normalerweise wird der Intervall-Wert als Angabe in Millisekunden aufgefasst, in der die Ressource ausgelesen wird. Allerdings sollte es auch möglich sein, diesen Wert zum Beispiel als Sekunden zu interpretieren. Dies wird durch das Element `<type>...</type>` bewerkstelligt, das als Wert *milliseconds* oder *seconds* unterstützt. Bei Bedarf lassen sich weitere Angaben integrieren. Zusätzlich kann man noch mit `<min_allowed>...</min_allowed>` und `<max_allowed>...</max_allowed>` einen minimalen und maximalen Schwellwert angeben, in dem sich das Intervall befinden muss. Damit kann zum Beispiel verhindert werden, dass ein Client mit einem sehr niedrigen Intervall den Server überlastet.

- `<thresholds>...</thresholds>`
  Für Push-Adapter wird außerdem noch die Möglichkeit angeboten, einen minimalen und maximalen Schwellwert für die eingelesenen Werte anzugeben. Falls der Wert unter oder über den Schwellwerten liegt, wird er einfach verworfen. Die dafür zuständigen Elemente sind wie auch bei Intervallen `<min_allowed>...</min_allowed>` und `<max_allowed>...</max_allowed>`.

- `<process>...</process>`
  Des weiteren muss ein Adapter wissen, welche Komponente der plattformabhängigen Schicht ausgeführt werden soll. Dies kann man mit dem Element `<process>...</process>` angeben. Darin zu verwendende Elemente sind `<cmd>...</cmd>`, welches den konkreten Aufruf enthält und `<cmd_params>...</cmd_params>`, innerhalb dem man mit `<param>...</param>` beliebig viele Parameter an den auszuführenden Prozess übergeben kann. Für `<cmd>...</cmd>` kann man zudem noch ein Attribut *external* angeben, welches Werte vom Typ *Boolean* annimmt. Wenn die Auswertung dieses Wertes *falsch* ergibt, dann wird die plattformabhängige Schicht umgangen.

Eine kurze Beispielkonfiguration für einen Pull-Adapter ist in Listing 3.1 zu sehen.

Listing 3.1: Beispielkonfiguration für einen Pull-Adapter

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <adapter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xsi:noNamespaceSchemaLocation="adapter_cfg.xsd"
4           id="SNMP␣TTL␣example">
5       <pull_adapter>
6           <source id="PullAdapter.A/LocalAdapters.AG/␣\
7  ␣␣␣␣␣␣␣␣␣␣␣pcheger12.HN/nm\.ifi\.lmu\.de.DN/SNMPTTL.S/int.ST">
8               <src_type>int</src_type>
9               <description>Pull TTL</description>
10              <interval>
11                  <type>milliseconds</type>
12                  <max_allowed>10000</max_allowed>
13                  <min_allowed>500</min_allowed>
14              </interval>
15              <process>
16                  <cmd>scripts/system_info/snmpget</cmd>
17                  <cmd_params>
18                     <param>public</param>
19                     <param>.1.3.6.1.2.1.4.2.0</param>
20                  </cmd_params>
21              </process>
22          </source>
23      </pull_adapter>
24 </adapter>
```

Abschließend stellt sich die Frage, wie die Korrektheit der Angaben in der Konfigurationsdatei überprüft werden kann. Dazu wird ein XML Schema definiert, das dem Parser übergeben wird und anhand derer er feststellen kann, ob die Angaben korrekt sind. Der Inhalt der Schema-Datei steht in Anhang A.

3.5. Zusammenfassung

In diesem Kapitel wurde ausführlich auf die Schnittstellen der plattformunabhängigen Schicht sowie der Integrations- und Konfigurationsschicht eingegangen. Ferner wurden Vor- und Nachteile der Verwendung der plattformabhängigen Schicht beim Zugriff auf Datenquellen erläutert. Größter Vorteil hierbei ist, dass der Ressourcenzugriff aus den Adaptern ausgelagert wird, so dass sehr leicht neue Datenquellen hinzugefügt werden können. Als Nachteil ist die geringere Leistung anzusehen. Danach wurde mit Hilfe eines Klassendiagramms die Struktur der Adaptoren veranschaulicht. Wichtig hierbei ist, dass Adaptoren mehrere Adapter-Datenquellen verwalten und die Adapter-Datenquellen für die Erzeugung von Monitor-Objekten sowie von Objekten, die die gemessenen Daten an die Integrations- und Konfigurationsschicht weiterleiten, verantwortlich sind. Schließlich wurde auf das Format von Konfigurationsdateien eingegangen, denen eine wesentliche Rolle bei der Erweiterung um neue Datenquellen zukommt.

4. Implementierung

Nachdem die wesentlichen Merkmale der Adaptoren im Entwurf behandelt wurden, sollen die dort gesammelten Erkenntnisse nun im Detail anhand von Quelltextbeispielen besprochen werden. Dazu werden die Aktivitäten der wichtigsten Anwendungsfälle beschrieben sowie Interaktionsdiagramme vorgestellt, um die Zusammenhänge der Objekte besser zu verstehen. Danach wird auf den Zugriff auf die Datenquellen eingegangen. Außerdem soll noch erklärt werden, wie die Erweiterung um neuen Datenquellen realisiert ist. Zuerst beschäftigen wir uns aber mit den Aktionen, die ein Adapter unmittelbar nach seinem Start ausführt. Dazu gehört das Einlesen der Konfigurationsdatei sowie die Initialisierung von CORBA, bei der die Servants, auf die die Integrations- und Konfigurationsschicht zugreifen, erzeugt werden, so dass man dann schließlich die Anwendungsfälle durchgehen kann.

4.1. Einlesen der Konfigurationsdatei

Gleich zu Beginn wird die Konfigurationsdatei, die wie in Kapitel 3.4 beschrieben aufgebaut ist, eingelesen. Alle für den Adapter allgemeine Angaben werden in einem Objekt vom Typ *AdapterConfiguration* gespeichert und alle für eine Adapter-Datenquelle relevanten Werte in einem Objekt vom Typ *SourceConfiguration*. Wie diese beiden Klassen zusammenhängen, wird in Abbildung 4.1 veranschaulicht.

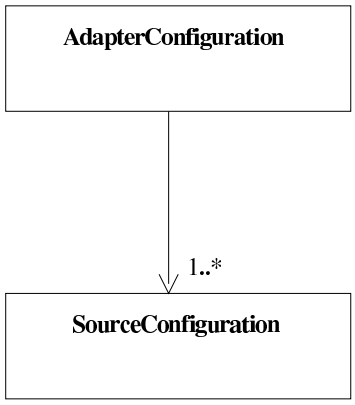

Abbildung 4.1.: Klassen, in denen die Konfiguration gespeichert wird

Zum Parsen der Konfigurationsdatei wurde eine eigene Parser-Klasse *XMLParser* geschrieben, die jedoch auf der Funktionalität der des Xerces C++ XML Parsers aufsetzt. Diese Abhängigkeit ist in Abbildung 4.2 dargestellt. Die Methoden *startElement()*, *endElement()* und *characters()* sind schon in *HandlerBase* definiert und als virtuell deklariert, müssen aber in *XMLParser* neu definiert werden. Darin stehen die Aktionen, die ausgeführt werden, wenn der Parser den Start eines Elements, das Ende eines Elements sowie Daten dazwischen ließt. Dieser Parsing-Prozess der Eingabedatei wird automatisch gemacht, *XMLParser* muss nur die Aktionen implementieren. Bei *startElement()* wird das durch *name* gegebene Element einem intern verwalteten Tag-Stack hinzugefügt, so dass *characters()* bestimmen kann, zu welchem Element die gelesenen Daten gehören. Außerdem werden in *startElement()* auch Elemente verarbeitet, die keine Daten enthalten oder aber Attribute beinhalten können. *endElement()* nimmt nur die entsprechenden Tags vom Tag-Stack. Dem Parser-Objekt vom Typ *XMLParser* wird eine Instanz von *AdapterConfiguration* übergeben, welche während dem Parsing-Prozess, der mit der Methode *parse()* gestartet wird, mit Werten von den Methoden *startElement()*

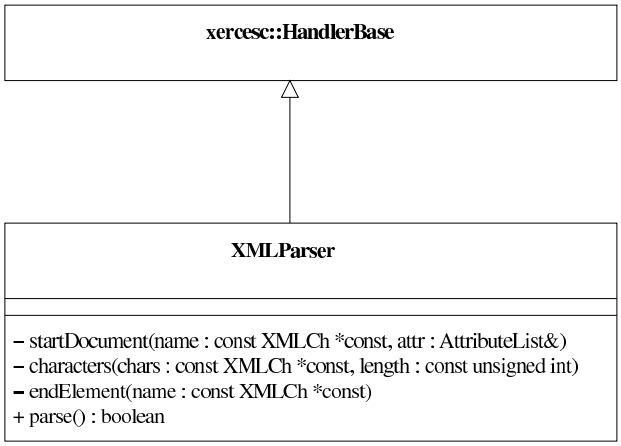

Abbildung 4.2.: Abhängigkeiten der Klasse XMLParser

und *characters()* aufgefüllt wird. Falls ein Fehler auftritt liefert *parse()* den Wert *false* zurück und der Adapter bricht mit einer Fehlermeldung ab (siehe Listing 4.1).

Listing 4.1: Starten des Parsing-Prozesses

```
1 AdapterConfiguration adapterConfig;
2 string xmlFile;
3 ...
4 XMLParser *parser = new XMLParser(xmlFile, adapterConfig);
5 if (!parser->parse()) {
6     cerr << "Error: cannot parse config file" << endl;
7     exit(1);
8 }
9 delete parser;
```

4.2. Initialisierung von CORBA

Zuerst muss der ORB initialisiert werden, was durch die Methode *ORB_init* erledigt wird. Dieser Methode wird ein Array von Strings sowie die Anzahl der Elemente in diesem Array übergeben. Wenn ein String mit *-ORB* beginnt, wird er als Parameter für den ORB aufgefasst. So kann man zum Beispiel mit der Option *-ORBendPoint* den Endpunkt der Kommunikation festlegen. Die Methode liefert ein Objekt zurück, welches den ORB repräsentiert. Alle weiteren Operationen müssen auf diesem Objekt aufgerufen werden.

```
static CORBA::ORB_ptr orb;
...
CORBA::Object_var obj;
PortableServer::ObjectId_var servantId;
PushAdapter_Impl *pushAdapter = 0;
PullAdapter_Impl *pullAdapter = 0;
...
orb = CORBA::ORB_init(argc, argv);
```

Als nächstes wird das Root-POA Objekt erzeugt. Da die Voreinstellungen ausreichend sind, müssen keine weiteren POAs erzeugt werden.

```
obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
```

Nun ist es an der Zeit, die Servants zu erstellen. Es werden momentan nur Servants für Push- und Pull-Adaptoren unterstützt. Den Servants wird eine Liste der zu unterstützenden Adapter-Datenquellen mitgegeben. Im Anschluss müssen die Servants beim Root-POA registriert und aktiviert werden.

```
if (adapterConfig.adapterType == PUSH_ADAPTER) {
    pushAdapter = new PushAdapter_Impl(adapterConfig.adapterId, version, \
            adapterConfig.sourceConfigList);
    servantId = rootPOA->activate_object(pushAdapter);
} else if (adapterConfig.adapterType == PULL_ADAPTER) {
    pullAdapter = new PullAdapter_Impl(adapterConfig.adapterId, version, \
            adapterConfig.sourceConfigList);
    servantId = rootPOA->activate_object(pullAdapter);
} else if (adapterConfig.adapterType == SET_ADAPTER) {
    cerr << "Error: set adapter not supported yet" << endl;
    CORBA::release(orb);
    exit(1);
}
```

Die zurückgelieferte Objekt-ID muss jetzt in eine Objektreferenz umgewandelt werden. Diese Objektreferenz wird danach beim Naming Service registriert. Der Client erhält bei einer Anfrage genau diese Objektreferenz.

```
obj = rootPOA->id_to_reference(servantId);
smona::SmonaAdapter_var objRef = smona::SmonaAdapter::_narrow(obj);
bindNames(objRef, adapterConfig.sourceConfigList);
```

Die POAs werden von dem POA-Manager verwaltet, der noch aktiviert werden muss.

```
PortableServer::POAManager_var poaMgr = rootPOA->the_POAManager();
poaMgr->activate();
```

Schließlich wird der Ausführungsfokus an den ORB übergeben. Von nun an werden eingehende Verbindungen vom ORB entgegengenommen und über den Root-POA an den zugehörigen Servant geleitet.

```
orb->run();
```

4.3. Implementierung der Servants

Nachdem der ORB bereit ist, Anfragen vom Client entgegenzunehmen, stellt sich die Frage, wie die Servants, an die die Anfragen weitergeleitet werden, auf diese reagieren. Darauf wollen wir anhand der in Kapitel 2.4 vorgestellten Anwendungsfälle näher eingehen, allerdings für Pull-Adaptoren. Zwar kommen Pull-Adaptoren in der Praxis seltener vor als Push-Adaptoren, da letztere dazu dienen, Ereignisse bei der Überschreitung von Schwellenwerten auszulösen, was für die Überwachung von Ressourcen von großer Bedeutung ist, allerdings sind Pull-Adaptoren aufgrund der Unterstützung von Timeouts, Historien und der asynchronen Behandlung von expliziten Pull-Anfragen in der Implementierung komplexer. Zuvor werden jedoch noch grundlegende Implementierungsaspekte der Servants erklärt.

4.3.1. Initialisierung der Servants

In Kapitel 4.2 wurde gezeigt, wie Instanzen eines Push- oder Pull-Servants in der *main()*-Methode erzeugt werden. Der Konstruktor der Servants ist dabei für die Initialisierung verantwortlich. Ein Argument des Konstruktors ist die Liste *sourceList* vom Typ *SourceConfigurationList* die aus den aus der Konfigurationsdatei eingelesenen Informationen über die Adapter-Datenquellen besteht. Jede dieser Adapter-Datenquellen wird von einem Objekt vom Typ *PushAdapterSource* oder *PullAdapterSource* repräsentiert. Da jede Adapter-Datenquelle eindeutig durch einen String identifizierbar ist, ist eine Zuordnung von diesem String zu einem

Adapter-Datenquellen-Objekt intern über die *sourceIdMap* vom Typ *std::map<std::string, AdapterSource *>*, wobei der Schlüssel die ID der Adapter-Datenquelle ist. Die Erzeugung der entsprechenden Objekte für einen Pull-Adapter erfolgt dann indem die Liste durchgelaufen wird und für den verwendeten Typ der entsprechende Template-Parameter für das Objekt angegeben wird (siehe Listing 4.2).

Listing 4.2: Initialisierung der Adapter-Datenquellen-Objekte

```
1  SourceConfigurationList::iterator p;
2  for (p = sourceList.begin(); p != sourceList.end(); p++) {
3      ...
4      } else if (adapterType == PULL_ADAPTER) {
5          switch (p->sourceType) {
6              ...
7              case LONG:
8                  sourceIdMap[p->sourceId] =
9                      new PullAdapterSource<CORBA::Long>(*p);
10                 break;
11             ...
12         }
13     }
14 }
```

Der Konstruktor von *PullAdapterSource* erhält also die Server-Konfiguration vom Typ *SourceConfiguration*, auf die intern über den Bezeichner *sourceConf* zugegriffen werden kann.

Die Servants sind jetzt vollständig bereit für Clientanfragen.

4.3.2. Allgemeines zu der Klasse AdapterSource

Abbildung 4.3 zeigt die Klasse *AdapterSource*. Aus Abbildung 3.5 ist ersichtlich, dass *AdapterSource* Basis-

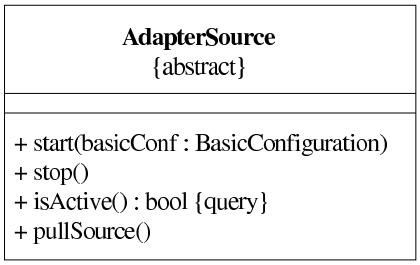

Abbildung 4.3.: Die abstrakte Klasse AdapterSource

klasse von *PushAdapterSource* und *PullAdapterSource* ist. Die Methoden *start()*, *stop()* und *isActive()* sind abstrakte, virtuelle Methoden, die von den abgeleiteten Klassen implementiert werden müssen. Dadurch wird ermöglicht, dass *SmonaAdapter_Impl* mittels dynamischer Bindung nur eine Datenstruktur zur Verwaltung der Adapter-Datenquellen-Objekte benötigt. Die Methode *pullSource()* ist nicht abstrakt, da sie nur von *PullAdapterSource* implementiert wird. Daher wäre es eigentlich auch sinnvoll, diese Methode nicht in die Schnittstelle von *AdapterSource* aufzunehmen. Dann müsste in *SmonaAdapter_Impl* jedoch explizit mittels der C++ Spracheigenschaft *dynamic_cast* die Schnittstelle von *PullAdapterSource* bekannt gemacht werden. Da es sich aber um eine Template-Klasse handelt, müsste man jedes mal den Typ überprüfen, weshalb es sinnvoller ist, die Methode in *AdapterSource* mit anzugeben. Die Methoden *start()*, *stop()* und *pullSource()* werden dann von den dazugehörigen Methoden aus *SmonaAdapter_Impl* aufgerufen.

PullAdapterSource dient des weiteren als Container für alle Threads, die zur Beantwortung der Clientanfragen notwendig sind. Da diese Threads Daten gemeinsam nutzen müssen, wird in *PullAdapterSource* eine Datenstruktur *SharedData* deklariert, in der alle wichtigen Einstellungen von *sourceConf* und *basicConf* gespeichert sind.

4.3.3. Wichtige Anwendungsfälle

Adapter Starten und Konfigurieren

Für diesen Anwendungsfall wird die Methode *start()* des Servants aufgerufen. Die Interaktionen werden in Abbildung 4.4 gezeigt. In Kapitel 3.3 ist bemerkt worden, dass die abstrakten Methoden in *SmonaAdapter*

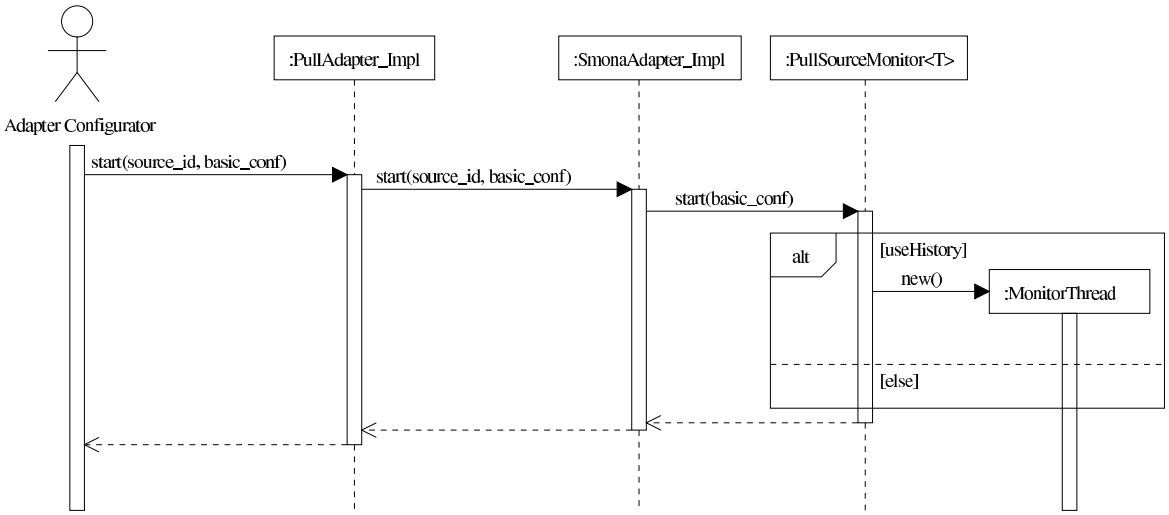

Abbildung 4.4.: Interaktionen der start()-Methode

in *PushAdapter_Impl* und *PullAdapter_Impl* nur als Wrappermethoden implementiert sind. Die eigentliche Implementierung befindet sich in *SmonaAdapter_Impl*. Auf diese Weise ist es möglich, Quellcode für Push- und Pull-Adaptoren gemeinsam zu nutzen. Die bei *PullAdapter_Impl* eingehende Anfrage an die Methode *start()* sieht folgendermaßen aus:

```
try {
    SmonaAdapter_Impl::start(source_id, basic_conf);
} catch (...) {
    throw;
}
```

Das Verhalten der *start()*-Methode in *SmonaAdapter_Impl* ist in Abbildung 4.5 dargestellt. Da CORBA bei mehreren Anfrage auch mehrere Threads erzeugt, ist es notwendig, eine Sperre anzufordern, um wechselseitigen Ausschluß zu realisieren. Danach wird überprüft, ob *source_id* in der verwendeten Map ein entsprechendes Adapter-Datenquellen-Objekt zugeordnet ist. Falls das nicht der Fall ist, wird diese Adapter-Datenquelle vom Adapter nicht unterstützt und es wird die Exception *NoSuchSourceException* geworfen. An dieser Stelle sollte angemerkt werden, dass die Adapter-Datenquellen-Objekte schon zu Beginn im Konstruktor von *SmonaAdapter_Impl* erzeugt werden (siehe Kapitel 4.3.1). Als nächstes wird ermittelt, ob die Adapter-Datenquelle schon aktiv ist, was dann der Fall ist, wenn zum einen schon die *start()*-Methode aufgerufen wurde und zum anderen die *stop()*-Methode noch nicht aufgerufen wurde und der Adapter keinen Fehler beim auslesen der Ressource entdeckt hat. Falls die Adapter-Datenquelle aktiv ist, wird eine Exception vom Typ *AdapterAlreadyActiveException* an den Client zurückgeliefert, ansonsten ruft der Adapter selbst *stop()* auf.

Danach wird die *start()*-Methode von *PullAdapterSource* aufgerufen, die zuerst ein neues Objekt *sharedData* vom Typ *SharedData* erzeugt, in dem alle wichtigen Informationen gespeichert werden.

```
sharedData = new SharedData;

sharedData->isActive = true;
sharedData->sourceId = sourceConf.sourceId;
...
```

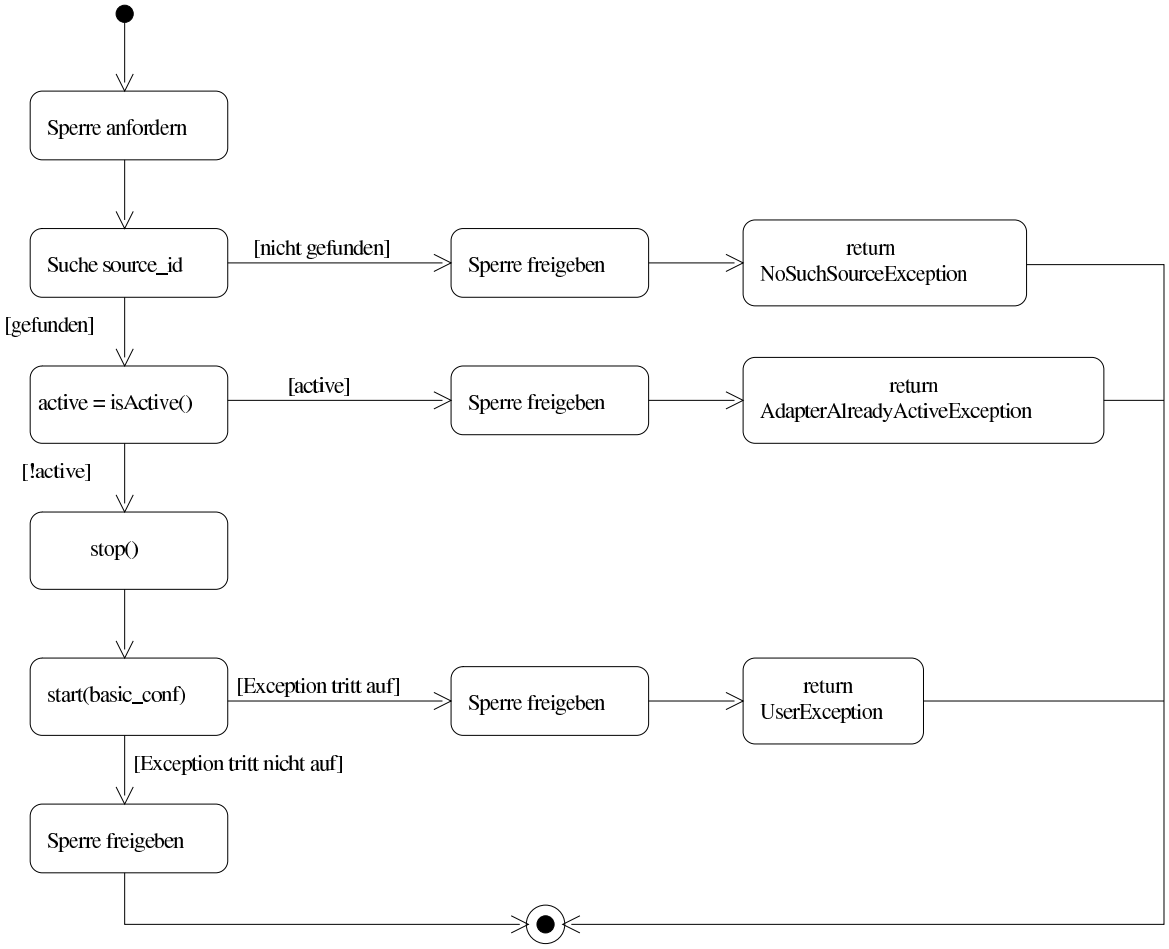

Abbildung 4.5.: Aktivitätsdiagramm der start()-Methode in SmonaAdapter_Impl

Schließlich werden die weiteren vom Client durch *basic_conf* übergebenen Konfigurationsparameter verarbeitet. Sollte dabei eine nicht unterstützte Eigenschaft angegeben sein, wird die Exception *NoSuchPropertyException* zurückgeliefert, im Fall eines ungültigen Wertes die Exception *InvalidPropertyValueException*. Zusätzlich kann es noch Eigenschaften geben, deren Angaben notwendig sind. Dazu gehört zum Beispiel die Eigenschaft, mit Hilfe derer der Adapter auf die Schnittstelle des Clients, also der Konfigurations- und Integrationsschicht, zugreifen kann. Hierbei wird ein Objekt vom Typ *PullListener* übertragen, welches auf Serverseite in dem Member *listener* der Klasse *SharedData* gespeichert wird. Falls solch eine Eigenschaft nicht in *basic_conf* steht, wird die Exception *MissingPropertyException* geworfen. Falls der Pull-Adapter eine Historie verwenden soll, wird zudem noch ein entsprechendes Monitor-Objekt erzeugt.

Die Aufteilung der *start()*-Methode auf zwei Objekte ist deshalb notwendig, da einige der Konfigurationsparameter vom Typ *CORBA::Any* sind, also erst einem konkreten Typ zugewiesen werden müssen, der aber vom Template-Parameter abhängig ist, wozu ein Objekt vom Typ *PullAdapterSource* benötigt wird.

Das fertiggestellte Objekt *sharedData* wird dann allen Threads übergeben und stellt einen von allen Threads gemeinsam genutzten Speicher dar.

Pull Source

Im Gegensatz zu *start()* ist die Methode *pullSource()* nicht in *SmonaAdapter_Impl* implementiert, sondern in *PullAdapter_Impl*, da sie nur für Pull-Adaptoren Sinn macht.

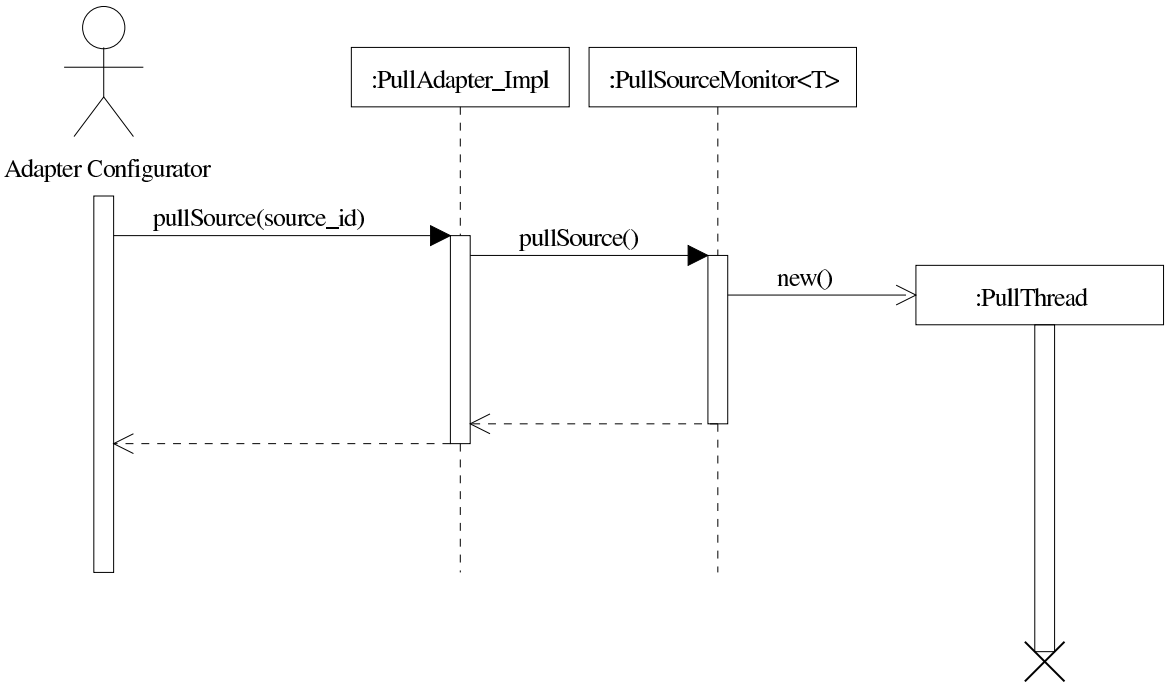

Abbildung 4.6.: Interaktionen der pullSource()-Methode

Nachdem *start()* aufgerufen wurde um die entsprechende Adapter-Datenquelle zu konfigurieren, kann der Client nun beliebig oft den Wert dieser Datenquelle mittels *pullSource()* abfragen. Zuerst wird, analog zu der Methode *start()*, überprüft, ob die ID der Adapter-Datenquelle gültig ist und das Adapter-Datenquellen-Objekt aktiv ist. Falls das alles zutrifft, wird die in *PullAdapterSource* definierte Methode *pullSource()* aufgerufen, ansonsten wird die Exception *NoSuchSourceException* bzw. *AdapterNotConfiguredException* zurückgeliefert.

Bei den Anforderungen wurde bereits erwähnt, dass eine Anfrage bei einer Datenquelle durchaus mehr Zeit in Anspruch nehmen kann, als der Client tolerieren will. Um den Client also nicht so lange zu blockieren, ist es notwendig, *pullSource()* asynchron zu implementieren. Um diese Asynchronität zu erreichen, wird bei

jeder Pull-Anfrage ein eigener Thread *pullThread* erzeugt, dem als Argument das in *start()* erzeugte Objekt *sharedData* übergeben wird:

```
pthread_t pullTID;
pthread_create(&pullTID, 0, &pullThread, static_cast<void *>(sharedData));
```

Danach kehrt die Methode *pullSource()* wieder zurück, der Client kann sich anderen Aufgaben zuwenden. Dies wird in Abbildung 4.6 verdeutlicht. Wenn der erzeugte Thread den Wert von der Datenquelle eingelesen und verarbeitet hat, greift er auf die vom Client angebotene Schnittstelle über *listener* zu, um den Wert an den Client zu senden. Nachfolgend soll der Pull-Thread im Detail erklärt werden. Hierbei werden drei Fälle unterschieden:

1. Ohne Historie, ohne Timeout
   Dies ist der Standard-Fall, bei dem über die *read()*-Methode von *SourceAccess* (siehe Kapitel 4.4) auf die Datenquelle zugegriffen und der gelesene Wert über die *send_Value()*-Methode von *PullListener* (siehe Kapitel 3.2.2) sofort an den Client weitergegeben wird. Der Quellcode dazu steht in Listing 4.3. Der Zugriff auf die Datenqulle kann beliebig lang dauern. Die Funtionen *stringToType()* und *typeToAny()* sind Hilfsfunktionen, wobei erstere den von der Datenquelle gelesenen Wert in den in der Adapter-Konfigurationsdatei angegebenen Typen konvertiert und letztere diesen Typen in das spezielle CORBA-Objekt *CORBA::Any* verpackt, welches dann in dieser Form übertragen wird. Der Client ist für das extrahieren des Wertes aus dem CORBA::Any-Objekt unter Bewahrung seines Typs verantwortlich. Diese Funktionen sind Template-Spezialisierungen und wenn neue Typen, also zum Beispiel strukturierte Datentypen, vom Adapter unterstützt werden sollen, müssen neue Spezialisierungen hinzugefügt werden.

Listing 4.3: pullSource() ohne Historie und ohne Timeout

```
1  SharedData *pullConf = static_cast<SharedData *>(arg);
2  string sourceId = pullConf->sourceId;
3  SourceAccess source(pullConf->cmd);
4  std::string value;
5  T result;
6  CORBA::Any toSend;
7  ...
8  source.read(value);
9  ...
10 HelperFunctions::stringToType<T>(value, result);
11 ...
12 HelperFunctions::typeToAny<T>(result, toSend);
13 (pullConf->listener)->
14     send_Value(CORBA::string_dup(sourceId.c_str()), toSend);
```

2. Ohne Histoire, mit Timeout
   Für den Client soll die Möglichkeit existieren, einen Wert anzugeben, der maximal vom Adapter für die Abfrage der Datenquelle benötigt werden darf. Da die Abfrage asynchron verläuft und daher der Client nicht geblockt wird, ist es somit auch nicht notwendig den Thread explizit zu beenden, wenn der Timeout abgelaufen ist, sondern es reicht aus dem Thread zu signalisieren, dass der Client keine Daten mehr erwartet und somit *send_Value()* nicht mehr aufgerufen werden soll.

   Der Timeout wird durch einen eigenen Timeout-Thread, der vom Pull-Thread gestartet wird, realisiert und dem ein Objekt vom Typ *Timeout*, das in Abbildung 4.7 dargestellt ist, übergeben wird. Über dieses Objekt können Pull-Thread und Timeout-Thread nun Informationen austauschen. Die Verarbeitung eines Timeouts geschieht dann folgendermaßen: Der Timeout-Thread blockiert sich selbst für die in *interval* angegebene Zeit unter Berücksichtigung von *intervalType*. Wenn diese Zeit abgelaufen ist, wird *timedOut* auf *true* gesetzt und ein Signal an den Pull-Thread mit Thread-ID *threadId* gesendet, um dessen Systemaufrufe abzubrechen, damit er möglichst schnell beendet werden kann. Da der Pull-Thread jedoch nur während dem Zugriff auf die Datenquelle blockieren kann wird dementsprechend davor und danach dieses Signal blockiert.

```
pthread_sigmask(SIG_UNBLOCK, &sigset, 0);
source.read(value);
pthread_sigmask(SIG_BLOCK, &sigset, 0);
```

Nach der Methode *read()* wird der Timeout-Thread beendet. Schließlich überprüft der Pull-Thread , ob *timedOut* gesetzt ist. Wenn dies der Fall ist, wird die Methode *send_Value()* nicht mehr aufgerufen.

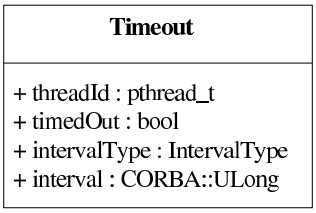

Abbildung 4.7.: Die Struktur Timeout

Da Signale verloren gehen können, ist schließlich noch eine Synchronisation zwischen Pull-Thread und Timeout-Thread notwendig:

```
if (!pullConf->useHistory && pullConf->interval > 0) {
    pthread_join(timeoutThreadId, 0);
}
```

3. Mit Historie
   Eine weitere Konifgurationsmöglichkeit für den Client besteht durch die Verwendung einer Historie. Der Zugriff auf die Datenquelle findet dann grundsätzlich nicht mehr in *pullSource()* statt, sondern in einem eigenen Monitor-Thread analog zu einem Push-Adapter, der in der *start()*-Methode erzeugt wird. Die nach jedem Interval gelesenen Werte werden in eine Datenstruktur geschrieben, die dann vom Pull-Thread ausgelesen werden kann. Der dafür zuständige Quellcode ist in Listing 4.4 gezeigt.

Listing 4.4: Auszug aus dem Monitor-Thread bei Verwendung einer Historie

```
1  try {
2      pthread_sigmask(SIG_UNBLOCK, &sigset, 0);
3      source.read(value);
4      pthread_sigmask(SIG_BLOCK, &sigset, 0);
5      T result;
6      HelperFunctions::stringToType<T>(value, result);
7      pthread_mutex_lock(&monitorConf->attrQueueMutex);
8      if ((monitorConf->attrQueue).size() >= monitorConf->maxElements)
9          (monitorConf->attrQueue).pop();
10     (monitorConf->attrQueue).push(result);
11     pthread_mutex_unlock(&monitorConf->attrQueueMutex);
12 }
```

Wenn die Datenstruktur leer ist, was meistens dann vorkommt, wenn *pullSource()* undmittelbar nach *start()* aufgerufen wurde, wird entsprechend eine Fehlermeldung an den Client gesendet (siehe Listing 4.5).

Listing 4.5: pullSource() bei Verwendung einer Historie

```
1      if (pullConf->useHistory) {
2          if ((pullConf->attrQueue).empty()) {
3              try {
4                  (pullConf->listener)->send_SourceError(CORBA::string_dup(sourceId.c_str()),
5                          CORBA::string_dup("No attribute available"), 0);
6              }
7              ...
8          } else {
```

```
 9              result = (pullConf->attrQueue).front();
10          }
11      }
```

Adapter Stoppen

Wenn eine Adapter-Datenquelle nicht mehr gebraucht wird, kann mit der *stop()*-Methode der Status des Adapter-Datenquellen-Objektes auf inaktiv zurückgesetzt werden. Danach sind keine Abfragen mit *pullSource()* mehr möglich. Allerdings ist auch die *stop()*-Methode asynchron, demnach findet keine Synchronisation mit den noch laufenden Pull-Threads statt, da *stop()* hierbei blocken könnte. Daher kann es vorkommen, dass, nachdem der Client *stop()* aufgerufen hat, immer noch Werte an den Client übermittelt werden. Die Alternative wäre, dass man die Pull-Threads explizit von der Methode *stop()* aus beendet, was aber zu einem Problem führt, wenn *start()*, *pullSource()* und *stop()* unmittelbar nacheinander ausgeführt werden, weil dann mit großer Wahrscheinlichkeit *pullSource()* seine Aufgabe nicht mehr erledigen kann.

4.4. Zugriff auf die Datenquellen

Als letztes soll noch der konkrete Zugriff auf die Datenquelle besprochen werden. Dazu erzeugt die plattformunabhängige Schicht ein Objekt vom Typ *SourceAccess*. Dieses bietet die Methode *read()* an, der als Argument ein Objekt vom Typ *SourceAccessData* übergeben wird, welches in Abbildung 4.8 dargestellt wird. Falls *ex-*

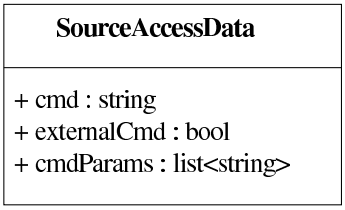

Abbildung 4.8.: Die Struktur SourceAccessData

ternalCmd den Wert *true* hat (dies entspricht der Grundeinstellung), enthält das Element *cmd* den Namen des auszuführenden Programms, welches in der Konfigurationsdatei angegeben wurde und der Komponente der plattformabhängigen Schicht entspricht. In diesem Fall wird aus dem Programmnamen *cmd* und den Parametern *cmdParams* der vollständige Befehl erstellt und unter dem Bezeichner *command* gespeichert. Dieser wird schließlich dazu benutzt, einen neuen Prozess zu erzeugen, der mit einer Pipe mit dem Adapter verbunden wird. Dieser neu erzeugte Prozess ist für den Ressourcenzugriff verantwortlich und schreibt das Ergebnis in die Pipe, wodurch die *read()*-Methode auf das Ergebnis zugreifen kann. Die *read()*-Methode liefert dieses schließlich durch das Argument *result* an die aufrufende Instanz zurück.

```
FILE *fp = 0;
if ( (fp = popen(command.c_str(), "r")) == 0) {
    ...
}
while (fgets(buf, sizeof(buf), fp) != 0) {
    result += buf;
}
```

In Listing 4.6 ist eine Komponente der plattformabhängigen Schicht gezeigt, die mittels dem *Simple Network Management Protocol* (SNMP) *get*-Aufrufe an den lokalen SNMP-Agenten stellt.

Listing 4.6: Skript zum Zugriff einen SNMP Agenten

```
1 #!/bin/bash
2
```

```
3  # SNMP get request
4  #
5  # Arguments:
6  #   COMMUNITY STRING
7  #   OBJECT IDENTIFIER
8  #
9  # Note: net-snmp and a running and properly configured snmp agent
10 #       are prerequisites for this script to work
11
12 if test $# -lt 2
13 then
14     echo Argument missing
15     exit 1
16 fi
17
18 x=`/usr/bin/snmpget -v1 -c $1 localhost $2 2>&1`
19
20 if test $? -ne 0
21 then
22     echo $x
23     exit 1
24 fi
25
26 echo $x |/bin/sed 's/.*::.*: //' 2>&1
27 exit $?
```

Eine Adapter-Datenquelle mit Unterstützung für SNMP ist insofern sehr interessant, dass damit ein Zugriff auf alle der in den von den Agenten unterstützten MIBs spezifizierten Netz- und Systemkomponenten möglich ist. Der Adapter kann der Integrations- und Konfigurationsschicht also eine große Anzahl an Attributen liefern, ohne sich speziell um den Zugriff auf die Datenquellen zu kümmern. Diese Attribute müssen natürlich noch zu geeigneten Dienstattributen zusammengefügt werden.

Nun wollen wir den Fall betrachten, dass *externalCmd* auf *false* gesetzt wurde. Dann wird adapterintern auf die Datenquellen zugegriffen. Dieser Zugriff erfolgt in einer für jede Datenquelle eigenen Klasse, die die Schnittstelle *DirectSourceAccess* implementiert. Diese Schnittstelle besitzt nur die Methode *execute()*. Schließlich werden in der *read()*-Methode Instanzen von diesen Klassen erzeugt. Der Bezeichner *cmd* wird dann dazu benutzt, die zugehörige Klasse auszuwählen. Dies wird in Abbildung 4.9 veranschaulicht. Die in der Abbildung

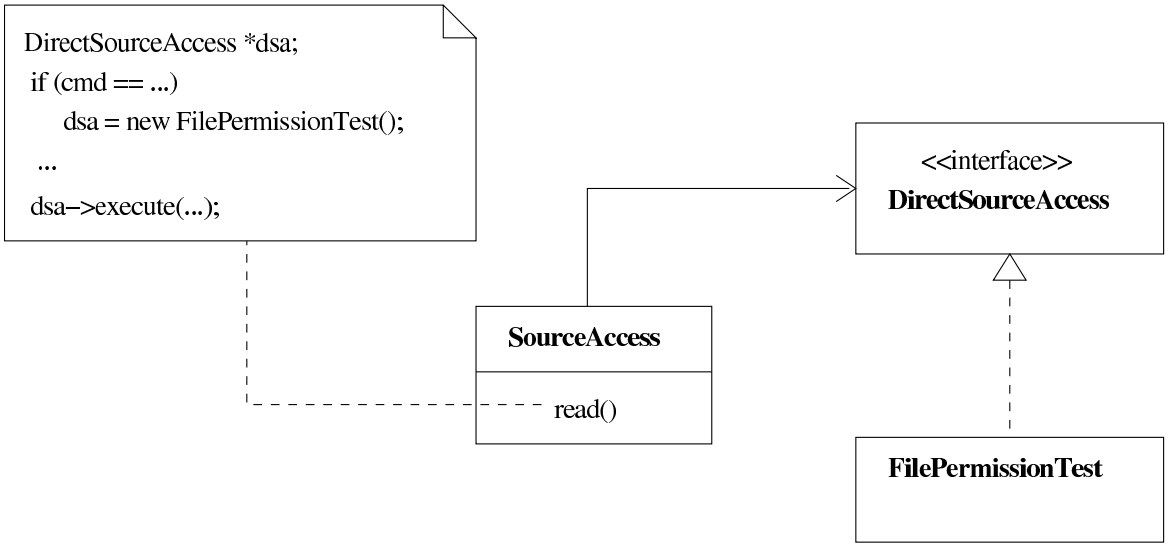

Abbildung 4.9.: Direkter Zugriff auf Datenquellen

gezeigte Beispielklasse *FilePermissionTest* überwacht die Zugriffsrechte für das Schreiben einer Datei unter UNIX/Linux. Die *execute()*-Methode kann dann zum Beispiel folgendermaßen implementiert werden:

```
struct stat statbuf;
if (stat((cmdParams.front()).c_str(), &statbuf) != 0) {
    ...
}

if (statbuf.st_mode & S_IWGRP || statbuf.st_mode & S_IWOTH)
    result += "true";
else
    result += "false";
```

Es soll also der Integrations- und Konfigurationsschicht mitgeteilt werden, wenn ein anderer als der Besitzer der Datei Schreibrechte erhält. Die dazugehörige Konfiguratiosndatei ist in Listing 4.7 dargestellt.

Listing 4.7: Konfigurationsdatei eines Push-Adapters für die Überwachung von Schreibrechten einer Datei

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <adapter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xsi:noNamespaceSchemaLocation="adapter_cfg.xsd"
4           id="Push␣adapter␣example">
5       <push_adapter>
6           <source id="PushAdapter.A/LocalAdapters.AG/pcheger12.HN/
7  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nm\.ifi\.lmu\.de.DN/FilePermissionTest.S/float.ST">
8               <src_type>boolean</src_type>
9               <description>Test whether group or other are allowed to write
10                 to a file</description>
11              <interval>
12                  <type>milliseconds</type>
13                  <max_allowed>60000</max_allowed>
14                  <min_allowed>100</min_allowed>
15              </interval>
16              <process>
17                  <cmd external="false">file_permission_test</cmd>
18                  <cmd_params>
19                     <param>/pfad/dateiname</param>
20                  </cmd_params>
21              </process>
22          </source>
23      </push_adapter>
24 </adapter>
```

4.5. Integration neuer Datenquellen

Um das Dienstmanagement in möglichst vielen Bereichen anwenden zu können, ist eine hohe Anzahl an zu unterstützenden Datenquellen nötig. Deshalb wurden die Adaptoren so entworfen, dass sich weitere Datenquellen leicht integrieren lassen, so dass nicht für jede Datenquelle ein eigener Adapter geschrieben werden muss. Dies wird zum einen durch die Ausgliederung des Ressourcenzugriffs auf die plattformabhängige Schicht über externe Programmaufrufe, die durch Konfigurationsdateien entsprechend konfiguriert werden, erreicht. Zum anderen erfordert die Integration einer neuen Datenquelle unter Umständen auch eine Erweiterung des Quellcodes der Adaptoren. Es wurde schon gesagt, dass *CORBA::Any*-Objekte zwischen der plattformunabhängigen Schicht und der Integrations- und Konfigurationsschicht übertragen werden und dass in diesen Objekten auch Informationen über den verwendeten Datentyp gespeichert werden. Es ist also notwendig, den durch den externen Programmaufruf erhaltenen Wert in einen Wert eines bestimmten Typs zu konvertieren und diesen Wert dann in einem *CORBA::Any*-Objekt zu speichern. Diese Aufgaben werden mit Hilfe von Template-Funktionen und deren Spezialisierungen erledigt. Nachfolgend wird die Konvertierung zu einem

double bzw. zu einem *string* gezeigt. Bei nicht Standard-Typen müssen weitere Spezialisierungen hinzugefügt werden.

```
/** Converts a string to a CORBA::Double.
 */
template<> void HelperFunctions::stringToType<CORBA::Double> \\
    (std::string data, CORBA::Double& to)
{
    to = strtod(data.c_str(), (char **)NULL);
}

/** Converts a string to a string.
 */
template<> void HelperFunctions::stringToType<std::string> \\
    (std::string data, std::string& to)
{
    to = data;
}
```

Bei der Speicherung in ein *CORBA::Any*-Objekt für die Übertragung tritt zudem ein weiteres Problem auf. Die Klasse *CORBA::Any* bietet den Operator <<= an, mit dem sich Werte einfügen lassen. Dieser Operator unterstützt jedoch nur die Standard-Typen und zusätzlich jeden in der IDL-Datei definierten Typ, da für diese der IDL-Compiler einen entsprechenden Operator hinzugefügt. Dementsprechend müssen also in der IDL-Datei nicht Standard-Typen hinzugefügt werden. Die Speicherung folgt dann auf gleiche Art und Weise wie bei den anderen Typen und es ist keine Anpassung bei den Template-Funktionen nötig.

```
template<class T> void HelperFunctions::typeToAny(T& type, CORBA::Any& any)
{
    any <<= type;
}
```

Es soll noch darauf hingewiesen werden, dass es davon auch CORBA-interne Ausnahmen gibt, wie zum Beispiel bei dem Typ *CORBA::Boolean*, für den folgende Template-Spezialisierung existiert.

```
/** Converts a boolean to a CORBA::Any.
 */
template<> void HelperFunctions::typeToAny<CORBA::Boolean> \\
    (CORBA::Boolean& type, CORBA::Any& any)
{
    any <<= CORBA::Any::from_boolean(type);
}
```

4.6. Zusammenfassung

Aus dem in Kapitel 3.3 vorgestellten Klassendiagramm wurden in diesem Kapitel für nicht-triviale Anwendungsfälle wie zum Beispiel Starten eines Pull-Adapters oder der Anfrage an einen Pull-Adapter, einen Wert zu messen, Sequenzdiagramme entwickelt. Das Verhalten der beteiligten Objekte sowie Operationen wurde daraufhin ausführlich mit Hilfe von Ausschnitten aus dem Quellcode gezeigt. Man unterscheidet zwei mögliche Zugriffe auf eine Datenquelle: Zum einen indirekt unter Beanspruchung der Dienste der plattformabhängigen Schicht, zum anderen direkt im Adapter über Systemaufrufe. Für beide Möglichkeiten werden Objekte der Klasse *SourceAccess* genutzt. Für ersteres wird ein externes Programm der plattformabhängigen Schicht aufgerufen, das seine Ausgabe in eine zuvor erstellte Pipe schreibt, aus der der Adapter die Daten lesen kann. Für letzteres muss eine neue Klasse geschrieben werden, die die Methode *execute()* implementiert. Diese wird dann von *read()* aufgerufen und liefert das Ergebnis zurück. Schließlich wurde noch gezeigt, dass die Erweiterung um neue Datenquellen im wesentlichen durch die Verwendung von Templates realisiert ist.

5. Installation und Beispiel

In diesem Kapitel soll zuerst die Installation der Adaptoren für SMONA beschrieben werden und danach die Funktionsweise anhand eines Beispiels konkret ersichtlich werden.

5.1. Installation des Adapters

Der Adapter wird mit den Quellen und dazugehörigen Skripten zur Installation als komprimierte Archivdatei mit dem Namen `smona-adapter-0.1.tar.bz2` geliefert. Die Quellen können nun mit dem Befehl

```
tar -xjf smona-adapter.tar.bz2
```

in das aktuelle Verzeichnis entpackt werden. Die Baumstruktur der kopierten Dateien sieht folgendermaßen aus:

```
smona-adapter-0.1
|-- AUTHORS
|-- COPYING
|-- ChangeLog
|-- INSTALL
|-- Makefile.am
|-- Makefile.in
|-- NEWS
|-- README
|-- aclocal.m4
|-- configure
|-- configure.ac
|-- depcomp
|-- etc
|   |-- Makefile.am
|   |-- Makefile.in
|   |-- adapter_cfg.xsd
|   |-- pull_adapter_cfg.xml
|   |-- pull_snmp_cfg.xml
|   |-- push_adapter_cfg.xml
|   `-- push_filepermissiontest_cfg.xml
|-- idl
|   |-- AdapterFunction.idl
|   |-- Makefile.am
|   |-- Makefile.in
|   |-- PullAdapter.idl
|   |-- PushAdapter.idl
|   |-- SetAdapter.idl
|   `-- SmonaAdapter.idl
|-- install-sh
|-- missing
|-- scripts
|   |-- Makefile.am
|   |-- Makefile.in
|   `-- system_info
|       |-- mounts
|       |-- snmpget
```

```
|       `-- uptime
|-- src
|   |-- AdapterFunctions.h
|   |-- AdapterSource.h
|   |-- CORBA.h
|   |-- DirectSourceAccess.h
|   |-- FilePermissionTest.cc
|   |-- FilePermissionTest.h
|   |-- HelperFunctions.h
|   |-- Makefile.am
|   |-- Makefile.in
|   |-- PullAdapterSource.h
|   |-- PullAdapter_Impl.cc
|   |-- PullAdapter_Impl.h
|   |-- PushAdapterSource.h
|   |-- PushAdapter_Impl.cc
|   |-- PushAdapter_Impl.h
|   |-- SmonaAdapter_Impl.cc
|   |-- SmonaAdapter_Impl.h
|   |-- SourceAccess.cc
|   |-- SourceAccess.h
|   |-- XMLParser.cc
|   |-- XMLParser.h
|   |-- client.cc
|   |-- main.cc
|   `-- main.h
|-- start-client
|-- start-names
`-- start-server
```

Kompiliert wird der Adapter mit:

```
./configure
make
```

Eine Installation ist mit

```
make install
```

möglich. Das Verzeichnis, in das der Adapter beim Aufruf von `make install` installiert werden soll, kann man dem `configure`-Skript mitteilen:

```
./configure --prefix=path
```

Um den Adapter erfolgreich kompilieren zu können, wird die Installation eines C++ Compilers vorausgesetzt. Empfohlen wird `gcc (g++)`[1], der mit Version `3.4` getestet wurde.

Zusätzlich wird noch eine Implementierung von CORBA sowie ein Parser zum Parsen von XML-Dateien benötigt. Der Adapter verwendet als ORB `omniORB`[2] sowie den Xerces C++ XML Parser[3], die mit Version `4.0.5` bzw. `2.7.0` getestet wurden.

Sollte `configure` einen Fehler melden, dass eines dieser beiden Pakete nicht gefunden werden konnte, liegt das vermutlich daran, dass sie in ein Verzeichnis installiert wurden, welches nicht untersucht wurde. Den Pfad zu diesem Verzeichnis kann man `configure` explizit mitteilen:

```
./configure --with-omniorb=/path/to/omniorb --with-xercesc=/path/to/xercesc
```

[1] http://gcc.gnu.org
[2] http://omniorb.sourceforge.net/index.html
[3] http://xml.apache.org/xerces-c/

In diesem Fall ist es auch wichtig, der Umgebungsvariable `LD_LIBRARY_PATH` das Verzeichnis hinzuzufügen, in dem sich die dazugehörigen Bibliotheken befinden, also zum Beispiel:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/omniorb/lib:/path/to/xercesc/lib
```

Alternativ kann auch der Pfad zu den Programmbibliotheken in `/etc/ld.so.conf` hinzugefügt werden. Danach muss zur Aktualisierung `ldconfig` ausgeführt werden.

Weitere Anweisungen sind der Datei `INSTALL` im Hauptverzeichnis zu entnehmen.

5.2. Beispiel

Als nächstes soll der Adapter gestartet werden und ein Client einige der hier besprochenen Funktionalitäten testen. Zuerst muss allerdings eine geeignete Adapter-Datenquelle integriert werden. Dazu soll die Beispielkonfiguration aus Kapitel 3.4 und das dazugehörige Skript, welches in 4.4 vorgestellt wurde, verwendet werden. Da dieses Skript eine Installation der SNMP-Implementierung *net-snmp*[4] voraussetzt, ist es möglicherweise notwendig, diese noch zu installieren. Schließlich muss dann noch der Agent gestartet werden (als *root*):

```
snmpd
```

Damit nun aber auf ein CORBA-Objekt überhaupt zugegriffen werden kann, muss auch noch der Name-Server gestartet werden, welcher in dem omniORB-Paket den Namen `omniNames` hat. Gestartet wird er, indem das Skript `start-names` aus dem Hauptverzeichnis aufgerufen wird. Dieses Skript steht in Listing 5.1 und muss gegebenfalls noch editiert werden. Zum einen wird der Pfad `OMNINAMESDIR` benötigt, unter dem `omniORB` installiert ist, zum anderen schreibt `omniNames` seine Einstellungen in eine Datei, welche in dem Verzeichnis `LOGDIR` gespeichert wird. Als Argument kann dem Skript eine beliebige Portnummer, an der der Name-Server auf eingehende Verbindungen warten soll, übergeben werden. Wenn dieses Argument nicht angegeben wird, wird die Portnummer `2809` verwendet.

Listing 5.1: Skript zum Starten des CORBA Namensdienstes

```
1  #!/bin/bash
2
3  OMNINAMESDIR=""
4  LOGDIR=""
5  PORT=2809
6
7  if test -z "$OMNINAMESDIR" || test -z "$LOGDIR"
8  then
9      echo OMNINAMESDIR and LOGDIR must be set
10     exit 1
11 fi
12
13 case $# in
14 1)    PORT=$1 ;;
15 *)    ;;
16 esac
17
18 if test ! -d "$LOGDIR"
19 then
20     install -d $LOGDIR
21 fi
22 if test "${LOGDIR}/omninames*"
23 then
24     rm -f ${LOGDIR}/omninames*
25 fi
26
```

[4] http://www.net-snmp.org

```
27 ${OMNINAMESDIR}/omniNames -start ${PORT} \\
28     -logdir ${LOGDIR} -errlog ${LOGDIR}/omniORB.errors 1>&2 &
```

Danach muss der Adapter und Client gestartet werden, wozu wiederum im Hauptverzeichnis Startskripte zu finden sind, nämlich `start-server` und `start-client`, die in Listings 5.2 und 5.3 abgebildet sind. Bei `start-client` werden mit `HOST` und `PORT` Hostnamen und Portnummer angegeben, unter denen sich der Name-Server befindet. Bei `start-server` kann zusätzlich noch `ENDPOINT_HOST` und `ENDPOINT_PORT` angegeben werden. Diese Informationen werden in die beim Name-Server registierten Objektreferenzen gespeichert.

Listing 5.2: Skript zum Starten des Servers

```
1  #!/bin/bash
2
3  if test $# -eq 0
4  then
5      echo usage: $0 [option] filename
6      exit 1
7  fi
8
9  # Default hostname, port and endpoint
10 HOST=localhost
11 PORT=2809
12 ENDPOINT_HOST=""
13 ENDPOINT_PORT=""
14
15 src/smona-adapter $@ -- -ORBendPoint giop:tcp:${ENDPOINT_HOST}:${ENDPOINT_PORT} \\
16     -ORBInitRef NameService=corbaname:iiop:${HOST}:${PORT}
```

Listing 5.3: Skript zum Starten des Clients

```
1  #!/bin/bash
2
3  # Default hostname, port
4  HOST=localhost
5  PORT=2809
6
7  case $# in
8  1)    ;;
9  2)    HOST=$2 ;;
10 3)    HOST=$2; PORT=$3 ;;
11 *)    echo usage: $0 filename [host] [port]; exit 1 ;;
12 esac
13
14 src/smona-adapter-client $1 -- \\
15     -ORBInitRef NameService=corbaname:iiop:${HOST}:${PORT}
```

Um eine Aufzählung der Parameter des Adapters zu erlangen, kann der Adapter mit der Option `--help` aufgerufen werden:

```
./start-server --help
```

Um also den Adapter über die Loopback-Schnittstelle des lokalen Rechners zu testen, muss folgendes im Hauptverzeichnis eingeben werden:

```
./start-names
./start-server etc/pull_snmp_cfg.xml
./start-client etc/pull_snmp_cfg.xml
```

Die Ausgabe des Clients ist in Listing 5.4 gegeben. Auf die Implementation des Clients wird nicht näher eingegangen, da es sich nur um eine Ansammlung von Testfällen handelt. Es gilt zu beachten, dass, wenn eine Historie verwendet wird, beim ersten Aufruf von *pullSource()* meistens noch kein Wert in der Historie gespeichert ist und demnach eine Fehlermeldung ausgegeben wird, dass noch kein Attribut vorhanden ist. Erst wenn *pullSource()* erneut aufgerufen wird, ist mittlerweile wahrscheinlich genügend Zeit vergangen, so dass Attribute vorliegen. Außerdem kann ein Adapter beliebig oft gestoppt werden.

Listing 5.4: Ausgabe des Clients

```
1  Test pull adapter:
2
3  Test 1: pull source without history and without timeout
4  64
5
6  Test 2: pull source without history but with timeout of 1000 ms
7
8  Test 3: pull source with history, interval = 1000 ms
9  Error: No attribute available
10 64
11
12 Test 4: pull source without starting adapter
13 CORBA user exception: AdapterNotConfiguredException
14
15 Test 5: start adapter twice
16 CORBA user exception: AdapterAlreadyActiveException
17
18 Test 6: stop adapter twice
19 64
```

6. Zusammenfassung

Aufgabe des Fortgeschrittenenpraktikums war, systemnahe Adaptoren für Datenquellen auf Linux-Systemen zu entwickeln. Diese Adaptoren stellen eine Teilschicht von SMONA (siehe Kapitel 2.1) dar. Zuerst wurden die Anforderungen an die Adaptoren gestgelegt. Zentrale Bedeutung dabei hatte die Spezifikation der Schnittstellen der mit den Adaptoren kommunizierenden Schichten, mit Hilfe derer die Adaptoren konfiguriert werden können und die Adaptoren Informationen an die Integrations- und Konfigurationsschicht liefern können. Eine weitere zentrale Anforderung war die einfache Integration neuer Datenquellen. Dies wurde durch die Unterstüzung von externen Programmaufrufen, die Verwendung von Template-Klassen sowie die Möglichkeit, Adapter-Datenquellen über XML-Dateien zu konfigurieren, erreicht. Im Kapitel über den Entwurf der Adaptoren wurde besonders auf die Spezifikation der Schnittstellen eingegangen sowie anhand eines Klassendiagramms die Architektur der Adaptoren gezeigt. Außerdem wurde noch die Funktionsweise der Middleware-Plattform CORBA beschrieben. Die Publizierung von Objektreferenzen beim Naming-Service ist hierbei von besonders großer Relevanz, weil dadurch dem Client auf einfache Weise ermöglicht wird, die Dienste des Servers in Anspruch zu nehmen. Abgeschlossen wurde das Kapitel mit einer Einführung in den Aufbau der Konfigurationsdateien. In Kapitel 4 wurde schließlich auf die Implementierung der Adaptoren anhand bedeutender Anwendungsfälle eines Pull-Adapters eingegangen. Zusätzlich zu Beispielen aus dem Quellcode wurde das Verhalten des Systems in diesen Anwendungsfällen mit Hilfe von Sequenz- und Aktivitätsdiagrammen erklärt. Es wurden auch unterschiedliche Konfigurationen besprochen, wie die Verwendung von Timeouts oder einer Historie. Durch die Verwendung der Sprache C++ ist es möglich, direkt auf das Betriebssystem durch Systemaufrufe zuzugreifen und insofern auch direkt auf Datenquellen. Dadurch wird die plattformabhängige Schicht umgangen, was zu einer erhöhten Leistung der Adaptoren führt. Die Spracheigenschaft der Templates erlauben zudem eine einfache Erweiterung der Adaptoren zur Untersützung von neuen Datenquellen. Zu tun bliebe noch, einen weiteren Adaptertyp, mit dem auch Änderungen im System erfolgen können, zuzulassen. Eine weitere Verbesserung könnte eine eigene Threadverwaltung darstellen, da die Adaptoren bei hoher Belastung viele Threads erzeugen müssen. Auch wurde in dem Fortgeschrittenenpraktikum der gesamte Sicherheitsaspekt nicht beachtet. Erweiterungen diesbezüglich, zum Beispiel die Authentifizierung von Benutzern der Adaptoren, wären in Zukunft sinnvoll.

A. XML Schema Definition

Dieses XML Schema (nach [Duerr 06]) dient dazu, die Korrektheit der Konfigurationsdateien zu überprüfen (siehe Kapitel 3.4).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A Pretty Simple Adapter Configuration Schema
        </xsd:documentation>
    </xsd:annotation>

    <!-- adapter and childs pull_adapter, push_adapter and set_adapter -->
    <xsd:element name="adapter">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:restriction base="xsd:anyType">
                    <xsd:choice>
                        <xsd:element name="pull_adapter"
                            type="pullAndSetAdapterType"/>
                        <xsd:element name="push_adapter"
                            type="pushAdapterType"/>
                        <xsd:element name="set_adapter"
                            type="pullAndSetAdapterType"/>
                    </xsd:choice>
                    <xsd:attribute name="id" type="xsd:string" use="required"/>
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>

    <!-- complex type for pull and set adapter types -->
    <xsd:complexType name="pullAndSetAdapterType">
        <xsd:complexContent>
            <xsd:restriction base="xsd:anyType">
                <xsd:sequence>
                    <xsd:element name="source" maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="src_type"
                                    type="sourceTypes"/>
                                <xsd:element name="description"
                                    type="xsd:string"/>
                                <xsd:element name="interval"
                                    type="intervalType"/>
                                <xsd:element name="process" type="processType"/>
                            </xsd:sequence>
                            <xsd:attribute name="id" type="xsd:string"
                                use="required"/>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:restriction>
```

```
        </xsd:complexContent>
    </xsd:complexType>

    <!-- complex push adapter type -->
    <xsd:complexType name="pushAdapterType">
        <xsd:complexContent>
            <xsd:restriction base="xsd:anyType">
                <xsd:sequence>
                    <xsd:element name="source" maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="src_type"
                                    type="sourceTypes"/>
                                <xsd:element name="description"
                                    type="xsd:string"/>
                                <xsd:element name="interval"
                                    type="intervalType"/>
                                <xsd:element name="thresholds"
                                    type="thresholdsType" minOccurs="0"/>
                                <xsd:element name="process" type="processType"/>
                            </xsd:sequence>
                            <xsd:attribute name="id" type="xsd:string"
                                use="required"/>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
            </xsd:restriction>
        </xsd:complexContent>
    </xsd:complexType>

    <!-- allowed source types for adapter -->
    <xsd:simpleType name="sourceTypes">
        <xsd:restriction base="xsd:normalizedString">
            <xsd:enumeration value="float"/>
            <xsd:enumeration value="int"/>
            <xsd:enumeration value="string"/>
            <xsd:enumeration value="long"/>
            <xsd:enumeration value="short"/>
            <xsd:enumeration value="double"/>
            <xsd:enumeration value="boolean"/>
        </xsd:restriction>
    </xsd:simpleType>

    <!-- childs of interval element -->
    <xsd:complexType name="intervalType">
        <xsd:sequence>
            <xsd:element name="type" type="intervalUnit"/>
            <xsd:element name="max_allowed" type="xsd:positiveInteger"
                minOccurs="0"/>
            <xsd:element name="min_allowed" type="xsd:nonNegativeInteger"
                minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <!-- restrictions for child type of interval element -->
    <xsd:simpleType name="intervalUnit">
        <xsd:restriction base="xsd:normalizedString">
            <xsd:enumeration value="milliseconds"/>
            <xsd:enumeration value="seconds"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
    <!-- childs of process element -->
    <xsd:complexType name="processType">
        <xsd:sequence>
            <xsd:element name="cmd" type="xsd:normalizedString"/>
            <xsd:element name="cmd_params" minOccurs="0">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="param" type="xsd:normalizedString"
                            maxOccurs="unbounded"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

    <!-- childs of threshold element -->
    <xsd:complexType name="thresholdsType">
        <xsd:sequence>
            <xsd:element name="max_allowed" type="xsd:decimal"/>
            <xsd:element name="min_allowed" type="xsd:decimal"/>
        </xsd:sequence>
        <!-- for push adapters with boolean threshold implement this
        <xsd:choice>
            <xsd:sequence>
                <xsd:element name="max_allowed" type="xsd:decimal"/>
                <xsd:element name="min_allowed" type="xsd:decimal"/>
            </xsd:sequence>
            <xsd:element name="trigger_bool" type="xsd:boolean"/>
        </xsd:choice>
        -->
    </xsd:complexType>
</xsd:schema>
```

Literaturverzeichnis

[BVD 01] G. BROSE, A. VOGEL, K. DUDDY: *Java Programming with CORBA: Advanced Techniques for Building Distributed Application*. John Wiley & Sons, 3. Auflage, 2001.

[DF 06] V. A. DANCIU, N. GENTSCHEN FELDE: *Supporting service management data composition in grid environments*, 2006. In *Proceedings of the 13th Annual Workshop of HP Open View University Association*, 2006, 277-282, Infonomics-Consulting, Hewlett-Packard, France, Mai, 2006.

[DFS 07] V. A. DANCIU, N. GENTSCHEN FELDE, M. SAILER: *Declarative Specification of Service Management Attributes*, 2007. In *Moving From Bits to Business Value: Proceedings of the 2007 Integrated Management Symposium*, 2007, IFIP/IEEE, München, Mai, 2007.

[DHSH 06] V. A. DANCIU, A. HANNEMANN, M. SAILER und H.-G. HEGERING: *IT Service Management: Getting the View*, 2006. In *Kern, E. M., Hegering, H.-G., und Brügge, B. (Hrsg): Managing Development and Application of Digital Technologies*, 2006, 110-130, Springer-Verlag München, Germany, Juni 2006.

[DS 05] V. A. DANCIU, M. SAILER: *A monitoring architecture supporting service management data composition*, 2005. In *Proceedings of the 12th Annual Workshop of HP Open View University Association*, 393-396, HP, Porto, Portugal, Juli, 2005.

[Duerr 06] DÜRR, M.: *Entwicklung von Adaptoren für Datenquellen auf Linux-Systemen*, 2006.

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[HV 99] M. HENNING, S. VINOSKI: *Advanced CORBA Programming with C++*. Addison Wesley, ISBN: 0-201-37927-9, 1999.

[ORB 07] D. GRISBY, S.-L. LO, D. RIDDOCH: *The omniORB version 4.1 User's Guide*, 2007, http://omniorb.sourceforge.net/omni41/omniORB/ .

[Str 97] STROUSTRUP, B.: *The C++ Programming Language*. Addison Wesley, ISBN: 0-201-88954-4, 3. Auflage, 1997.

[XER 05] FOUNDATION, THE APACHE SOFTWARE: *Xerces-C++ Documentation*, 2005, http://xml.apache.org/xerces-c/pdf/xerces-c.pdf .

[XML 04] W3C, WORLD WIDE WEB CONSORTIUM: *Extensible Markup Language (XML) 1.1*, 2004, http://www.w3.org/TR/2004/REC-xml11-20040204/ .

[XS 04] W3C, WORLD WIDE WEB CONSORTIUM: *XML Schema Part 0: Primer Second Edition*, 2004, http://www.w3.org/TR/xmlschema-0/ .