8. Datenverwaltung in Amica

8.1. Entfernte Datenobjekte

In rechenintensiven verteilten Applikationen können gerade zwischen einzelnen Berechnungsphasen große Datenmengen anfallen. Diese passen häufig nicht in den Arbeitsspeicher eines kleineren Rechners und sollten auch möglichst selten über ein Netzwerk übertragen werden, um die Netzlast zu minimieren. Daher müssen sie entsprechend von der Infrastruktur behandelt werden und sind in der Applikationsarchitektur explizit zu spezifizieren. In dem hier präsentierten Erweiterungsmodul `RemoteDataObjects` werden Komponententypen zur Verfügung gestellt, die entfernte Datenobjekte repräsentieren. Die Schnittstellen, über die auf den Inhalt eines Datenobjekts zugegriffen werden kann, hängen von dem Typ des Datenobjekts ab.

Der Applikationsentwickler erzeugt ein Datenobjekt und verbindet es mit den Rechenkomponenten, die auf dieses zugreifen sollen. Zur Laufzeit sorgt dann eine Infrastruktur dafür, dass das Datenobjekt an einem Ort erzeugt wird, der genügend Platz hat und sich bzgl. der Zugriffsgeschwindigkeit des Netzwerks nahe an den Berechnungen befindet, die auf die Daten zugreifen.

Die Datentypen, die unterstützt werden, sind in dem Erweiterungsmodul `DataTypes` enthalten. Obwohl das Typkonzept sehr allgemein und offen angelegt ist, hat sich in den Fallstudien überraschenderweise gezeigt, dass wenige Typen ausreichen. Tabelle 8.1 zeigt die aktuell unterstützten Datentypen, die als neuer Typ `AmicaDataType` für Eigenschaften von Architekturelementen eingeführt wurde. Neben den konventionellen Typen, auf deren Beschreibung hier verzichtet wird, sind die Typen `Picture` und `Executable` auffällig. Der erste repräsentiert ein zweidimensionales farbiges Bild. Über dessen Schnittstelle ist der Zugriff auf Pixel und Bildausschnitte möglich und es lässt sich das Bild in verschiedene Bildformate exportieren. Der Typ `Executable` wird für ausführbaren Code eingesetzt. Eine detaillierte Beschreibung ist in Kapitel 10.2 zu finden.

8.1.1. Vokabular für Datenobjekte

Datenobjekte einer Applikation sind Komponenten vom Typ `DataObject`. Der Zugriff auf Datenobjekte von Rechenkomponenten geschieht über Konnektoren des Typs `DataAccess`. Abbildung 8.1 zeigt diese Klassen und die Zuordnung zu ihren grafischen Repräsentationen im Editor.

Datenobjekte haben einen Namen und eine Umgebung, in der dieser Name gültig ist. Jede Applikation hat eine eigene Umgebung, die automatisch als

Typ	Beschreibung
Binary	Folge von Bytes mit veränderlicher Größe
String	Zeichenkette
Picture	Farbiges Bild
Executable	Ausführbare Subapplikation, evtl. mit Quellcode
Array[Byte]	Folge von Bytes mit fester Größe
Map[Binary]	Abbildung von Zeichenketten auf Folgen von Bytes
Map[String]	Abbildung von Zeichenketten auf Zeichenketten

Tabelle 8.1.: Liste der unterstützten Datentypen

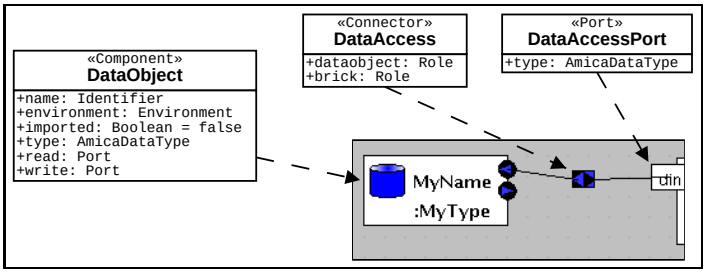

Abbildung 8.1.: Architekturelemente für Datenobjekte in Amica

Präfix vor die Umgebung des Datenobjekts gehängt wird. Dadurch können gleichzeitig Applikationen laufen, die Datenobjekte mit dem gleichen Namen benutzen. Möchte man auf externe Datenobjekte zugreifen, so ist das Attribut `imported` auf `true` zu setzen. In dem Fall findet die automatische Erweiterung nicht statt.

Zusätzlich besitzt ein Datenobjekt einen der schon beschriebenen Typen, der dessen Schnittstelle definiert und zwei Anschlüsse für einen lesenden und einen schreibenden Zugriff. Ist eine Rechenkomponente mit dem Anschluss für den lesenden Zugriff verbunden, so darf sie nur die Methoden aus der Schnittstelle des Datenobjekts benutzen, die keine Veränderung bei dem Datenobjekt hervorrufen. Ist sie mit dem Anschluss für schreibenden Zugriff verbunden, so darf sie nur verändernde Methoden benutzen. Ist sie mit beiden verbunden, steht ihr die gesamte Schnittstelle zur Verfügung. Diese Trennung der Zugriffsarten ermöglicht eine Analyse des Zugriffsverhalten von Rechenkomponenten und dient als Grundlage für Optimierung und Verifikation. Bis auf ein paar einfache Versuche wurden hierfür allerdings keine weiteren Untersuchungen durchgeführt.

Komponenten, die auf Datenobjekte zugreifen wollen, müssen hierfür einen speziellen Anschluss vom Typ `DataAccessPort` besitzen. Dieser kann dann über gerichtete `DataAccess` Konnektoren mit den Datenobjekten verbunden werden. Die Beschriftung `din` des Anschlusses in der Abbildung 8.1 ist sein willkürlich gewählter Name. Dieser Name ermöglicht es, eine Rechenkomponente unabhängig von bestimmten Datenobjekten zu spezifizieren, indem sie sich auf den Anschluss statt auf ein bestimmtes Datenobjekt bezieht.

8.1.2. Laufzeitunterstützung und Infrastruktur

Zur Laufzeit werden `DataObject`-Komponenten der Applikationsarchitektur durch CORBA Objekte repräsentiert. Die CORBA-Schnittstelle des Objekts wird anhand des spezifizierten Datentyps der Applikationskomponente bestimmt. Abbildung 8.2 zeigt einen Ausschnitt der Schnittstelle eines Datenobjekts vom Typ `Picture` in der Schnittstellenbeschreibungssprache IDL von CORBA. Über diese Schnittstelle kann ein Applikationsentwickler direkt auf ein Datenobjekt zugreifen.

```
interface PictureDO : DataObject
{
  attribute PictureSize Size;
  void setPixel(in unsigned long x,
                in unsigned long y,
                in Pixel val) raises (PixelIndexOutOfRange);
  Pixel getPixel(in unsigned long x,in unsigned long y);
  string getAsPPM3();
  OctetSeq getAsRaw();
  ULongSeq getAreaAsJDK(in unsigned long x1,
                        in unsigned long y1,
                        in unsigned long x2,
                        in unsigned long y2);
  void setAreaByJDK(in ULongSeq area,
                    in unsigned long x1,
                    in unsigned long y1,
                    in unsigned long width);
};
```

Abbildung 8.2.: Ausschnitt der Schnittstellenbeschreibung für ein `Picture` Datenobjekt

Um Kommunikationsaufwand zu reduzieren, sind Datenobjekte stets bzgl. der Übertragungsleistung im Netzwerk in der Nähe der Berechnungen positioniert. Dies kann z.B. ein Rechner in dem gleichen Netzwerk sein, mit dem auch die Rechner verbunden sind, welche die Berechnungen durchführen. Da mehrere Berechnungen auch parallel an unterschiedlichen Orten durchgeführt werden können, die auf die gleichen Daten zugreifen, ist es möglich, Datenobjekte zu replizieren. Bei dem Anlegen eines neuen Replikats muss der gesamte Inhalt des Datenobjekts über das Netzwerk übertragen werden. Da diese Datenmenge sehr groß sein kann, sollten besondere Eigenschaften des Netzwerks ausgenutzt werden, sofern dies möglich ist. Steht z.B. eine ATM-Verbindung zur Verfügung, sollte die Datenübertragung direkt unter Einsatz des ATM-Protokolls durchgeführt werden und nicht unter Einsatz höherer Übertragungsprotokolle, die auf dem ATM-Protokoll aufgesetzt sind. Auch wenn keine speziellen Übertragungsprotokolle zur Verfügung stehen, zeigten Messungen, dass durch die direkte Nutzung von TCP/IP eine um bis zu 15% höhere Übertragungsleistung im Vergleich zu IIOP[1] erzielt werden konn-

[1] Internet Inter-ORB Protocol: Ein auf TCP/IP aufbauendes Protokoll für die Kommunikation

te [FGWE98, Wol98].

Abbildung 8.3 gibt einen schematischen Überblick über die Infrastruktur. Datenobjekte werden von *Datenspeichern* verwaltet, die Zugriff auf die konkreten Speicherressourcen haben. Sie können das nächstgelegene Replikat eines Datenobjekts anhand dessen Namens lokalisieren. Neue Replikate werden ebenfalls von Datenspeichern angelegt. Zur besseren Ausnutzung des Netzwerks bei der Übertragung des Inhalts eines ganzen Datenobjekts werden *Verbindungsobjekte* benutzt.

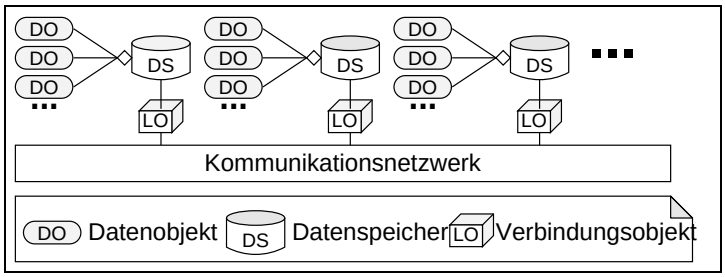

Abbildung 8.3.: Schematische Darstellung der Infrastruktur für entfernte replizierte Datenobjekte

Abbildung 8.4 zeigt ein Klassendiagramm der wichtigsten Klassen der Implementierung dieser Infrastruktur. Um ein Datenobjekt *o* zu lokalisieren, wendet sich der Benutzer mit dem Namen von *o* an einen prinzipiell beliebigen Datenspeicher *DS*. Datenspeicher sind Objekte der Klasse `DataStore`, die über CORBA fernaufrufbar sind. Die Lokalisierung des Datenobjekts geschieht über die Methode `getDataObject()`.

Besitzt *DS* ein Replikat von *o* liefert er eine entsprechende CORBA-Referenz zurück. Ansonsten leitet es die Frage an alle bekannten Datenspeicher weiter. Wird dort ein Replikat gefunden, wird dieses geliefert. Alternativ lässt sich auch bei *DS* ein neues Replikat von *o* anlegen, auf welches eine Referenz als Ergebnis zurückgegeben wird. Es gibt zwei Möglichkeiten, wie das Anlegen eines neues Replikats angestoßen werden kann. Erstens kann sich der Datenspeicher nach einer vorgegebenen Politik selbst für das Anlegen eines Replikats bei sich entscheiden. Zweitens kann der Benutzer, bzw. die von ihm gestartete Applikation, die Wissen über ihre weiteren Berechnungen hat, explizit ein Replikat anlegen.

Auf eine genaue Beschreibung des Replikationsvorgangs soll hier verzichtet werden. Verkürzt dargestellt, werden zunächst alle Replikate eines Datenobjekts nach einem Abgleich ihres Inhalts gesperrt. Daraufhin wird der Inhalt eines Replikats zu dem Datenspeicher übertragen, in dem das neue Replikat liegen soll. Das Replikat wird erzeugt, mit den anderen Replikaten vernetzt und schließlich werden alle Replikate entsperrt.

Für die Übertragung eines Replikats gibt es ein zentrales Objekt der Klasse `BindingController`[2], das in der Methode `transferData()` die Übertra-

mit CORBA-Objekten.

[2]Ein zentraler Ansatz führt natürlich schnell zu Problemen der Skalierbarkeit und Verfügbar-

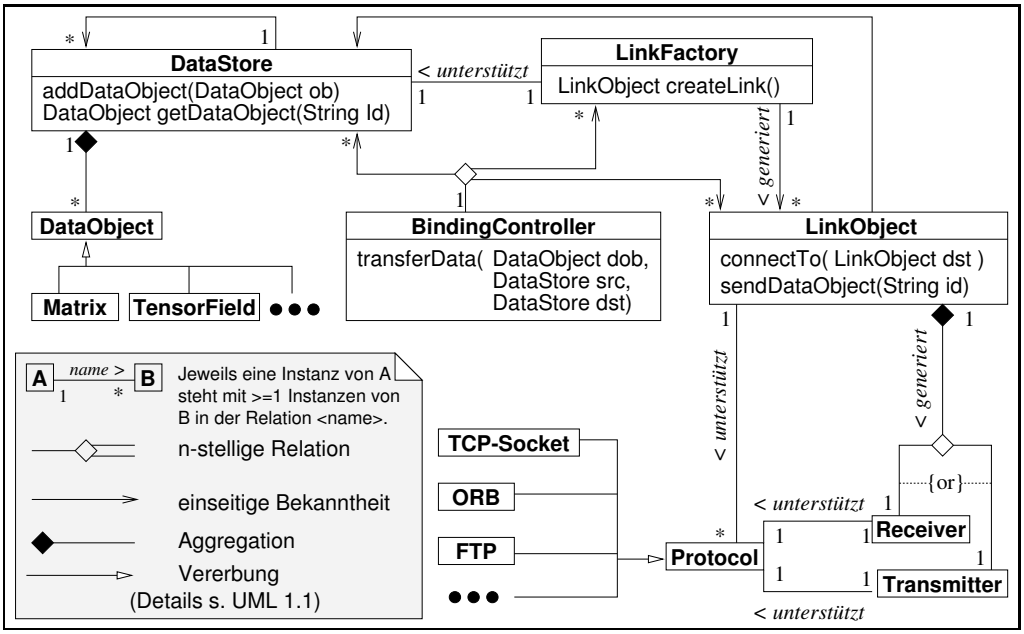

Abbildung 8.4.: Überblick über das Speichersubsystem

gung durchführt. Die eigentliche Datenübertragung wird von Objekten der Klasse `LinkObject` durchgeführt, die bei den Datenspeichern liegen und auf bestimmte Netzwerkprotokolle spezialisiert sind. Sie werden dynamisch von dem `BindingController`-Objekt über ein `LinkFactory`-Objekt, das ebenfalls bei dem zugehörigen Datenspeicher liegt, erzeugt, wenn sie benötigt werden.

Zusätzlich zu der Effizienzsteigerung durch den Einsatz spezifischer Protokolle ermöglicht dieser Ansatz der spezialisierten Verbindungsobjekte auch eine verschlüsselte Übertragung der Datenobjekte. Entscheidet sich der Benutzer, dass der Inhalt eines Datenobjektes vertraulich ist, markiert er dieses entsprechend. Ein derartig markiertes Datenobjekt wird von den Verbindungsobjekten für die Übertragung verschlüsselt. Zur Verwaltung der Schlüssel wurde eine verteilte Infrastruktur in prototypischer Form auf Grundlage der Java Cryptography Extension realisiert. Für weitere Details siehe [FKE99, Kra99].

Der konkrete Zugriff auf ein Datenobjekt geschieht über dessen CORBA-Schnittstellen. Neben der adäquaten Unterstützung komplexerer Datentypen erleichtert der Einsatz von CORBA die Implementierung aktiver Replikation der Datenobjekte. Eine Operation auf einem Datenobjekt von einer Applikation wird damit auf einem Replikat initiiert und dann an alle restlichen Replikate weitergeleitet. Konkret wird eine Operation zuerst auf dem lokalen Replikat ausgeführt und dann an eine Warteschlange angehängt. Ein eigener

keit. Daher sollte die Infrastruktur in Domänen unterteilt werden, wobei jede Domäne ihr eigenes `BindingController`-Objekt besitzt. Da die Erstellung einer weltweit skalierbaren Infrastruktur nicht das Hauptziel dieser Arbeit war, wurde auf eine Implementierung verzichtet.

Thread arbeitet diese Warteschlange ab, indem er ihr die jeweils erste Operation entnimmt und über CORBA auf allen entfernten Replikaten ausführt. Dabei wird erst dann die nächste wartende Operation bearbeitet, wenn keine entfernte Operationsausführung mehr aussteht.

Somit unterstützt der Replikationsmechanismus PRAM-Konsistenz[3], d.h. alle Schreiboperationen, die auf einem Replikat initiiert wurden, werden auf allen anderen Replikaten in der gleichen Reihenfolge ausgeführt [LS88]. Operationen, die auf unterschiedlichen Replikaten initiiert wurden, können sich aber beliebig vermischen.

Um dieses formal auszudrücken, seien alle Operationen, die auf einem Replikat ausgeführt werden, durch ihre Ausführungsreihenfolge total geordnet. Dies ist in dem Amica-System möglich, da die Operationen einen Leser-Schreiber-Ausschluss unterstützen, also auf einem Replikat zu einer Zeit höchstens eine Schreiboperation ausgeführt werden kann. Nebenläufige Lesezugriffe sind zwar möglich, aber da sie keinen Effekt auf das Replikat haben, werden sie im folgenden vernachlässigt. Sei $P_i^{\alpha,\beta}$ eine Operation P, die von dem Replikat α initiiert und auf dem Replikat β als i-te Operation in dessen Ausführungsreihenfolge ausgeführt wurde. Dann ist die folgende Eigenschaft in diesem Replikationsmechanismus stets gewährleistet:

$$(\forall\beta)((\exists P_i^{\alpha,\beta}, Q_j^{\alpha,\beta} : i < j) \rightarrow (\exists P_{\tilde{i}}^{\alpha,\alpha}, Q_{\tilde{j}}^{\alpha,\alpha} : \tilde{i} < \tilde{j}))$$

Obwohl die PRAM-Konsistenz vergleichsweise schwach ist, hat sie sich häufig als ausreichend erwiesen, da meistens zwischen den verteilten Applikationsfragmenten ein Erzeuger/Verbraucher-Kommunikationsmuster existiert. Einige Applikationselemente haben Zwischenergebnisse berechnet, welche dann von anderen als Eingabe benutzt wurden. In diesem Falle ist die PRAM-Konsistenz ausreichend.

Weisen Applikationsfragmente nun ein Kommunikationsverhalten auf, welches bei PRAM-Konsistenz zu einer inakzeptablen Inkonsistenz des Datenobjekts führen kann, müssen sie vor jeder Schreiboperation das Datenobjekt sperren. Hierfür steht eine zusätzliche Sperrsynchronisation zur Verfügung, die auf dem verteilten Algorithmus für einen gegenseitigen Ausschluss von Lamport basiert [Lam78].

Dieser Mechanismus erlaubt es, ein repliziertes Datenobjekt über genau ein Replikatobjekt zu modifizieren. Alle anderen Replikatobjekte sind gesperrt, d.h., alle Operationen blockieren solange, bis das Datenobjekt wieder entsperrt ist. Bei der Errichtung der Sperre, werden zuerst alle noch ausstehenden Aktualisierungsoperationen ausgeführt, aber keine neuen mehr angenommen. Eine Sperre wird erst dann wieder entfernt, wenn alle Aktualisierungsoperationen ausgeführt wurden, die von dem Replikat initiiert wurden, welches

[3]Die Abkürzung PRAM steht für *Pipelined RAM* und hat ihren Ursprung in der Anwendung als Konsistenzprotokoll für Caches oder für virtuellen gemeinsamen Speicher. Dabei wird bei Schreibzugriffen nicht auf die Aktualisierung aller Replikate gewartet, sondern nach der lokalen Aktualisierung direkt weiter gearbeitet. Wenn ein Prozess mehrere Schreibzugriffe sequenziell ausführt, werden diese somit im Sinne des Pipeline-Paradigma nebenläufig abgearbeitet.

Besitzer der Sperre war. Hiermit ist dann sequenzielle Konsistenz gewährleistet.

Die Entscheidung, ob PRAM-Konsistenz ausreichend ist, liegt bei dem Applikationsentwickler, da nur er die Anforderungen der Applikationslogik kennt. Enthält eine Applikation also Code, der direkt auf Datenobjekte zugreift, muss der Applikationsentwickler die Konsistenzeigenschaften der Datenobjekte verstehen.

Übersetzung und Laufzeitunterstützung

Datenobjektkomponenten der Applikationsarchitektur werden zu einfachen Applikationsobjekten übersetzt, die alle Informationen über das Datenobjekt, wie dessen Namen, Typ, etc., enthalten. Wird die Laufzeitunterstützung für Datenobjekte von dem Koordinator gestartet, so durchsucht sie alle Applikationsobjekte nach Datenobjektapplikationsobjekten. Die gefundenen werden intern gespeichert.

Alle Applikationsobjekte, deren Rechenkomponente über einen `DataAccessPort`-Anschluss mit einer Datenobjektkomponente verbunden sind, erhalten bei der Übersetzung den Namen des Datenobjekts. Benötigen sie nun zur Laufzeit Zugriff auf das Datenobjekt, so wenden sie sich mit dessen Namen an die Laufzeitunterstützung für Datenobjekte. Diese greift auf den nächstgelegenen Datenspeicher zu und erhält so eine CORBA-Referenz auf das Datenobjekt, die es an das anfragende Applikationsobjekt zurückgibt. Die Laufzeitunterstützung merkt sich dabei alle CORBA-Referenzen, um bei der nächsten Anfrage nach dem Datenobjekt den Aufruf des Datenspeichers zu vermeiden.

8.2. Datenflussparadigma

Eine Alternative zu dem Einsatz entfernter Datenobjekte, die dem Variablenmodell in imperativen Programmiersprachen entsprechen, sind Datenströme, die in Datenflusssprachen, wie z.B. Sisal [Sis02, Ray00], oder auch bei den Pipes von Unix Verwendung finden. Berechnungen erzeugen Daten, diese werden über einen Kommunikationskanal an andere Berechnungen gesendet, welche diese Daten dann verbrauchen. Dieser Ansatz hat zwei Vorteile:

- Die Berechnungen für die Erzeugung und für den Verbrauch der Daten können nebenläufig durchgeführt werden. Nachdem das erste erzeugte Datum bei den verbrauchenden Berechnungen angekommen ist, können diese beginnen.

- Es ist nicht nötig, alle erzeugten Daten zu lagern. Im Extremfall muss der Kanal nur eine Referenz auf das erzeugte Datum an den Verbraucher weitergeben und selber keinen Speicherplatz für das Datum selber zur Verfügung stellen. Im allgemeinen wird der Kanal allerdings einen kleinen Puffer bereitstellen, um Erzeuger und Verbraucher zu entkoppeln und so die potenzielle Nebenläufigkeit zu erhöhen.

Der Verbraucher hat bei diesem Ansatz allerdings keinen selektiven Zugriff auf die Daten. Wenn er einen solchen benötigt, muss er alle eingehenden Daten lokal bei sich puffern und diesen Puffer dann bearbeiten, nachdem er ausreichend gefüllt wurde. Zusätzlich sind die über den Kanal übertragenen Daten flüchtig, da sie nach dem Abliefern beim Verbraucher nicht nochmals verwendet werden können.

Sie stellen damit zwar keinen Ersatz für die im vorherigen Kapitel beschriebenen Datenobjekte dar sondern eine sinnvolle Ergänzung. Abbildung 8.5 zeigt die Elementtypen mit ihrer grafischen Darstellung in dem Editor. Das zentrale Element des Vokabulars ist der `StreamChannel`. Dieser stellt einen aktiven Kanal dar, der Datenfolgen von Erzeugern zu Verbrauchern sendet. Für Rechenkomponenten, die Daten erzeugen, steht der Anschlusstyp `StreamOut` zur Verfügung. Über diesen werden sie mit dem Kanal verbunden, der diese Daten dann weiterleitet. Entsprechend wird der Anschlusstyp `StreamIn` für den Empfang von Daten benutzt.

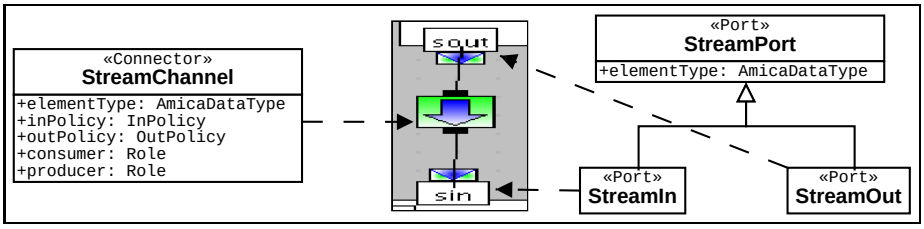

Abbildung 8.5.: Architekturelemente für Datenströme in Amica

Die Daten, die über einen Kanal transportiert werden, sind typisiert, wobei das gleiche Typsystem wie bei den Datenobjekten benutzt wird. Das Attribut `elementType` muss für einen Kanal stets den gleichen Wert besitzen, wie das gleichnamige Attribut in allen `StreamIn`- und `StreamOut`-Anschlüssen, mit denen der Kanal verbunden ist. Die Rolle `consumer` eines Kanals kann mit beliebig vielen `StreamIn`Anschlüssen verbunden werden, während ein `StreamIn` Anschluss mit höchstens einer `consumer`-Rolle verbunden sein kann. Das gleiche gilt für `StreamOut`-Anschlüsse und `producer`-Rollen.

Ein Kanal transportiert Datenfolgen, die mit einem ausgezeichneten Signal terminiert werden. Er kann beliebig viele eingehende Datenfolgen von produzierenden Rechenkomponenten verarbeiten und Datenfolgen für die mit ihm verbundenen verbrauchenden Rechenkomponenten erzeugen. Nach welchen Strategien er dabei vorgeht, wird über die Attribute `inPolicy` und `outPolicy` spezifiziert. In der aktuellen Version werden zwei Werte für das Attribut `inPolicy` und damit zwei Strategien für die Kombination eingehender Datenfolgen unterstützt.

- *Direkte Vermischung:*
  Jedes eingehende Datenelement wird unverzüglich zum Ausgabestrom weitergesendet. Dadurch werden eingehende Datenfolgen in Abhängigkeit ihrer Ankunftszeiten miteinander vermischt. Der Ausgabestrom terminiert erst dann, wenn alle Eingabefolgen terminiert sind.

Diese Strategie ist z.B. dann einsetzbar, wenn jedes einzelne Datenelement einen autonomen Auftrag für andere Rechenkomponenten darstellt. So können z.B. Überwachungskameras Datenströme erzeugen, die Bilderfolgen sind. Eine Rechenkomponente analysiert nun jedes Bild bzgl. gegebener Kriterien, wie. z.B. Rauchentwicklung oder unbefugtes Eindringen, und aktiviert gegebenenfalls einen globalen Alarm. In diesem Fall würden die Datenfolgen aller Kameras einfach vermischt und in die Analysekomponenten geleitet werden.

- *Multiplexen:*
  Eine eingehende Datenfolge wird vollständig als Ausgabedatenfolge versendet. Wenn sie terminiert, terminiert auch die Ausgabedatenfolge. Alle weiteren eingehenden Datenfolgen müssen auf diese Terminierung warten. Sie werden dann in der Reihenfolge ihres Eintreffens bei dem Strömungskonnektor aktiviert.

  Dieser Strategie eignet sich, wenn eine gesamte Eingabefolge eine autonome Aufgabe definiert. Wenn z.B. in dem obigen Beispiel eine Kamera nicht eine Folge von Bildern sondern eine Folge von Pixelfolgen senden würde, wobei jede Pixelfolge ein Bild darstellt, dann würde die Multiplexstrategie das gleiche Verhalten wie in dem obigen Beispiel die Vermischungsstrategie bewirken.

Die folgenden drei Strategien werden unterstützt, um die erzeugte Ausgabedatenfolge an die angeschlossenen Verbraucher zu verteilen.

- *Rundsendung:*
  Jedes Datenelement des Ausgabestroms wird an alle verbundenen Verbraucher geschickt. Dies stellt ein übliches Kommunikationsparadigma für verteilte Applikationen dar.

- *Elementweises Round-Robin*
  Diese Strategie erzeugt eine oder mehrere Ausgabedatenfolgen, indem die einzelnen Datenelemente aus den Eingabefolgen nacheinander elementweise auf die Ausgabefolgen verteilt werden. Genauer ausgedrückt, wird bei n Konsumenten für jeden Konsumenten k mit $0 \leq k < n$ eine Ausgabedatenfolge O_k erzeugt. Dabei werden die Datenelemente von O_k aus einer von der Eingabestrategie des Strömungskonentors erzeugten Ausgabedatenfolge $I = [d_0, d_1, d_2 \ldots d_{\nu-1}]$ nach dem Round-Robin Modell gewählt, so dass gilt:

  $$O_k = [d_{\rho_k(\xi)} \mid 0 \leq \xi < \lfloor \frac{\nu}{n} \rfloor + (\textbf{if } (k < \nu \bmod n) \textbf{ then } 1 \textbf{ else } 0)]$$

  mit

  $$\rho_k(\xi) = n\xi + k$$

  .
  Wenn I keine endliche Länge hat, dann sind auch alle O_k unendlich lang. Diese Strategie eignet sich, wenn die einzelnen Datenelemente autonome Aufträge darstellen, die zur Lastverteilung an unterschiedliche Rechenkomponenten gesendet werden.

- *Stromweises Round-Robin:*
  Im Gegensatz zu dem elementweisen Round-Robin wird hier eine gesamte von der Eingabestrategie erzeugte Ausgabefolge an einen Verbraucher geschickt. Die nächste Ausgabedatenfolge wird dann an den nächsten Verbraucher geschickt und so weiter. Diese Strategie eignet sich in Kombination mit der Multiplexen-Strategie für die Eingabefolgen um eine Lastverteilung zu erreichen.

Es sind natürlich weitere Strategien zur Kombination der Eingabedatenfolgen und zur Verteilung an die Konsumenten denkbar, aber diese fünf Strategien decken die üblichen Paradigmen paralleler Programme ab und haben sich bei den praktischen Fallstudien als ausreichend erwiesen.

Bei der Übersetzung einer Applikationsarchitektur wird für jeden `StreamChannel`-Konnektor ein einfaches Applikationsobjekt erzeugt, welches als Behälter für die Eigenschaften des Kanals dient, wie Typ, Verhaltensstrategien und der Name des Konnektors. Wird die Applikation gestartet, sucht die Laufzeitunterstützung für Datenströme nach diesen Applikationsobjekten und speichert Verweise auf sie.

Rechenkomponenten erhalten ähnlich wie bei den Datenobjekten bei der Übersetzung den Namen des Kanals, mit dem sie verbunden sind. Zur Laufzeit erhalten sie eine Referenz auf einen Kanal, indem sie die Laufzeitunterstützung für Datenströme kontaktieren und ihr den Kanalnamen übergeben. Falls der Kanal noch nicht existiert, erzeugt ihn die Laufzeitunterstützung mit den Informationen, die in dem entsprechenden Applikationsobjekt gespeichert ist.

Die Implementierung von Datenströmen basiert auf CORBA. Es existieren CORBA-Schnittstellen für Erzeuger, Verbraucher und Kanäle. Da CORBA keine generischen Typen besitzt, gibt es für jeden Datentyp **T**, der in einem Kanal übertragen wird, eine Familie von CORBA-Schnittstellen. Abbildung 8.6 zeigt die Familie für Zeichenketten, also mit **T**=`String`, mit den grundlegenden Basisklassen.

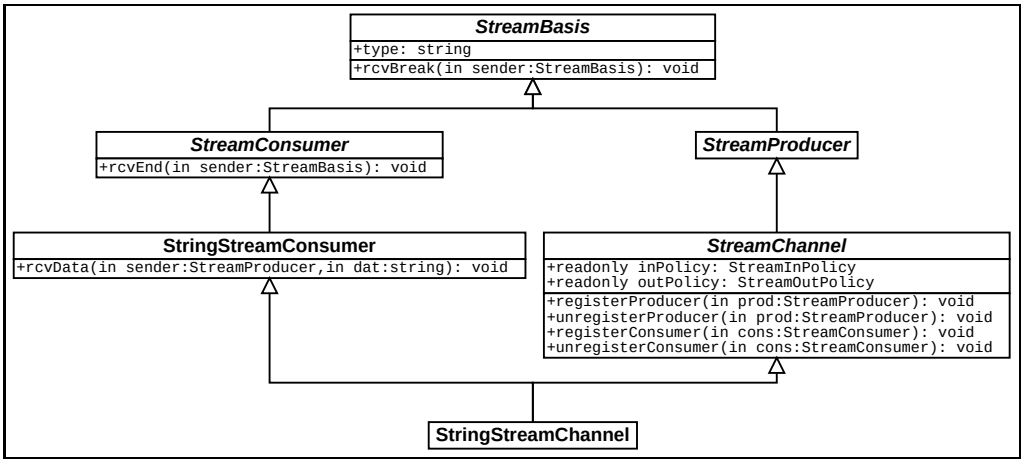

Abbildung 8.6.: Entfernter Zugriff auf Datenströme über CORBA

Wie aus den Klassen ersichtlich ist, wird nur ein *Push* Modell unterstützt. Das bedeutet, dass Erzeuger ihre Daten immer selbstständig an die Verbraucher senden. Es ist den Verbrauchern nicht möglich Daten abzuholen. Die nächsten Daten einer Folge werden erst dann zu einem Verbraucher verschickt, wenn dieser das vorherige Datum angenommen hat. Um eine höhere Effizienz zu erzielen, enthalten `StreamChannel`-Objekte interne Puffer zur Zwischenspeicherung empfangener Daten.

Zusätzlich zu dem Verschicken von Daten mit `rcvData()` und der Signalisierung des Endes einer Datenfolge mit `rcvEnd()` unterstützt die Infrastruktur auch noch mit `rcvBreak()` ein Signal, dass die Datenfolge zusammengebrochen ist. Wenn ein Kanal dieses Signal erhält, leitet er es an alle registrierten Erzeuger und Verbraucher weiter. Wie diese darauf reagieren, ist nicht allgemein spezifiziert, da es von ihrer konkreten Funktionalität abhängt, ob sie mit ihrer Tätigkeit fortfahren können oder abbrechen müssen. Wenn sie abbrechen, geben sie ein entsprechendes Signal an alle anderen Kanäle weiter, mit denen sie verbunden sind.

Wie schon erwähnt, werden Kanäle dynamisch zur Laufzeit erzeugt, wenn sie benötigt werden. Dabei werden sie entsprechend der Vorgaben aus der Applikationsarchitektur bzgl. der Verarbeitung ihrer Ein- und Ausgabefolgen parametrisiert. Diese Parametrisierung lässt sich nachträglich nicht mehr ändern. Kanäle werden von den Datenspeichern erzeugt, die auch die Datenobjekte verwalten. Dies ermöglicht es, für die Wahl des geeigneten Ortes für einen neu zu erzeugenden Kanal die gleichen Optimierungsmechanismen einzusetzen wie bei der Wahl eines Ortes für ein Datenobjekt. Die Implementierung der Verbraucher und Erzeuger ist unterschiedlich bei den einzelnen Typen von Rechenkomponenten. Sie wird in den entsprechenden Kapiteln erläutert.