10. Amica-Fallstudien

Um die Anwendbarkeit des Amica-Systems für rechenintensive verteilte Applikationen zu untersuchen, wurden mehrere Fallstudien durchgeführt, die unterschiedliche Aspekte des Systems nutzen. In Abschnitt 10.1 werden verteilte Simulationen zellulärer Systeme beschrieben. Dabei wurde die Infrastruktur um einen sehr konkreten Rechendienst, nämlich die Simulation selber, erweitert. Die Berechnungen liefen mit realen Umgebungsdaten, die 18 MB umfassten und damit die prinzipielle Brauchbarkeit der Architektur und der verwendeten Technologien Java und CORBA prüften.

In Abschnitt 10.2 wird die Infrastruktur um einen sehr generischen Dienst erweitert, der es ermöglicht Komponenten einzusetzen, die parallele Applikationen kapseln, welche für lokal verteilte parallele Systeme, wie z.B. Hochleistungsparallelrechner oder Workstation-Cluster, entworfen wurden. Dabei wird exemplarisch die in diesem Bereich sehr verbreitete nachrichtenbasierte Middleware MPI[1] unterstützt. In der letzten Applikation wird in Abschnitt 10.3 der Einsatz von Strömungskonnektoren untersucht.

Für die Fallstudien wurden die Amica-spezifischen Erweiterungsmodule gebündelt in einem ECL-Erweiterungsmodul implementiert. Die hier vorgestellten Auswertungen und Ergebnisse sind teilweise auch in [FK00, Fin00, FK01] veröffentlicht worden.

10.1. Simulation von Algorithmen zur Ressourcenverteilung in zellulären Systemen

Das Mobilfunksystemen zur Verfügung stehende Frequenzspektrum ist für die heutigen Anforderungen nur knapp ausreichend. Daher ist die Untersuchung von Algorithmen zur effizienten Verwaltung und Kontrolle dieser Ressource wichtig für Netzwerkbetreiber. Das gesamte Spektrum wird dabei üblicherweise in Kanäle unterteilt, welche die mobilen Geräte mit den Basisstationen der Netzwerkbetreiber verbinden. Wenn ein Anwender mit seinem Mobilfunkgerät eine Netzwerkzelle betritt oder verlässt, wird ihm ein solcher Kanal zugeteilt. Die Zuteilung von Kanälen zu Netzwerkzellen geschieht häufig statisch[2] und passt sich damit nur ungenügend an die hohe Dynamik in echten Systemen an, die von dem Anwenderverhalten, der zur Verfügung stehenden Ressourcen und weiterer externer Einflüsse, wie z.B. dem Wetter,

[1] Message Passing Interface [MF94]

[2] Diese Aussage galt zumindestens im Jahr 2000, zu dem Zeitpunkt, an dem diese Fallstudie durchgeführt wurde.

abhängt. Einen breiten Überblick über unterschiedliche Kanalvergabestrategien gibt [KN96].

Im Rahmen einer Kooperation der Friedrich-Alexander-Universität von Erlangen und der Forschungsgruppe für verbindungsfreie Systeme von Lucent Technologies im Projekt MONA[3] wurde ein Simulationswerkzeug namens *Steam* eingesetzt für die Bewertung von Kanalvergabestrategien. Es sind allerdings viele Simulationsläufe notwendig und jeder Simulationslauf benötigt mehrere Megabyte an Umgebungsdaten und eine starke Rechenleistung. Daher wurde untersucht, ob der Einsatz eines Systems wie Amica sich für eine effiziente Durchführung der Simulationen eignet.

Die Struktur der Applikation ist dabei sehr einfach und typisch für viele rechenintensive Applikationen die in weit verteilten Netzen eingesetzt werden. Der Simulator benötigt zwei Eingabedateien. Eine enthält die Umgebungsdaten und ist sehr umfangreich, die andere enthält einen Parametersatz. In der verteilten Applikation werden zuerst die Umgebungsdaten und alle Parametersätze geladen. Für jeden Parametersatz wird nebenläufig ein Simulationslauf unter Einsatz in dem verteilten System verfügbarer Rechenressourcen durchgeführt. Die Ergebnisse bestehen aus Protokolldateien, die dann an einem zentralen Punkte gespeichert und später außerhalb der Applikation ausgewertet werden. Der Grad erreichter Parallelität hängt von der Menge gleichzeitig verfügbarer Rechenressourcen ab.

Abbildung 10.1 zeigt die hierfür eingesetzte Applikationsarchitektur. Zuerst werden von zwei Adapterkomponenten zwei Dateien des lokalen Dateisystems des Rechners geladen, welcher den Koordinator für diese Applikation ausführt. Eine Datei enthält die Umgebungsdaten. Diese wird mit einer `JavaOperation`-Komponente `ReadBinary`, also einer allgemeinen parametrisierbaren Java-Klasse, die in einer Bibliothek zur Verfügung steht, in das Datenobjekt `SimData` geladen. Die dafür nötigen Parameter werden als Eigenschaften der Komponente spezifiziert.

Die zweite Datei enthält die Parametersätze. Sie besteht aus einer einfachen Textdatei, in der jede Zeile einen Parametersatz enthält. Für jeden Parametersatz wird ein Eintrag in dem Datenobjekt `paras` angelegt. Der hierfür erforderlicher Code musste vom Applikationsentwickler geschrieben werden.

Nachdem beide Adapterkomponenten ihre Aufgabe erledigt haben, wird die Farm-Komponente aktiviert. Diese erzeugt für jeden Eintrag in `paras` ein Arbeiterexemplar, das jeweils nur aus einer Dienstkomponente besteht. Der dabei angeforderte Dienst ist die Nutzung des oben beschriebenen Simulationswerkzeugs *Steam*. Alle Ergebnisse werden in dem Datenobjekt `Results` gespeichert. Abschließend werden sie von einer weiteren Adapterkomponente im lokalen Dateisystem gespeichert.

Da dieser Simulationsdienst ein sehr spezieller Dienst ist, der i.a. nicht a priori vorhanden ist, musste ebenfalls der Applikationsentwickler die Infrastruktur einmalig erweitern. Die Dienstausführung besteht dabei hauptsächlich aus einer parametrisierten Programmausführung. Da es möglich war, sich auf bestehende Klassen abzustützen, war die Einbettung des

[3] Mobile Network Analysis and Optimization

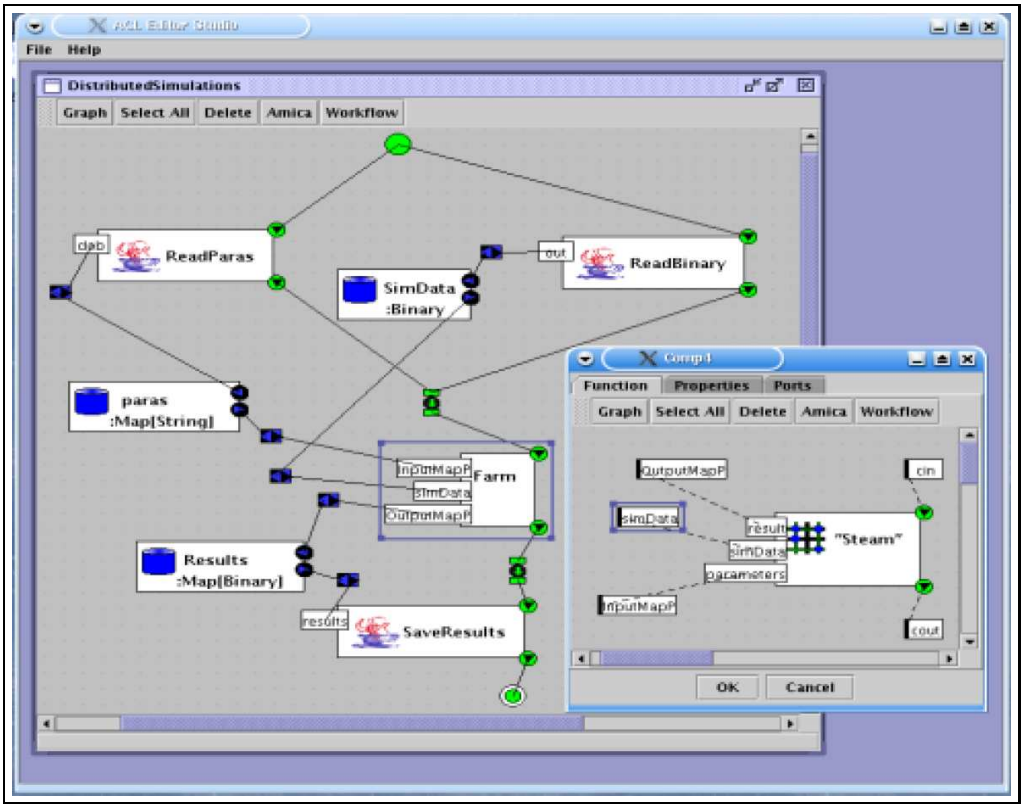

Abbildung 10.1.: Applikationsgraph für verteilte Simulationen von Kanalvergabestrategien von Mobilfunknetzen

Dienstes in die Infrastruktur einfach. Als einfaches Maß für die Bewertung der Güte eines Dienstanbieters wurde die Länge der Warteschlange benutzt. Ein genaueres Maß wäre aber unter Berücksichtigung der Simulationsparameter möglich. Es wurden Dienstanbieter für Sun-Solaris und für Linux Plattformen entwickelt.

Um nun den durch den Einsatz der komplexen Infrastruktur verursachten zusätzlichen Verwaltungs- und Berechnungsaufwand experimentell zu bestimmen, wurde die Laufzeit für eine variierende Anzahl von Rechnern gemessen. Eine homogene Auswahl von Sun-Rechnern[4] vereinfachte dabei die Interpretation der Ergebnisse. Diese waren sowohl über ein lokales Ethernet Netzwerk mit 10MBit als auch über das leistungsfähige Weitverkehrsnetzwerk DFN miteinander verbunden. Insgesamt wurden 10 Rechner[5] eingesetzt.

Die einzelnen Simulationen sind voneinander unabhängig. Sie kommunizieren nicht miteinander und bis auf den gemeinsamen Start und dem Warten auf die Terminierung aller Simulationen gibt es keine Synchronisation. Die maximal mögliche Beschleunigung S_{max} bei n nebenläufigen Simulationen bei einem Einsatz von p Prozessoren beträgt damit:

$$S_{max} = \frac{n}{\lceil n/p \rceil}$$

In dem Experiment wurden 10 gleiche Parametersätze für die Simulationen eingesetzt. Dabei wurde ein realistischer Umgebungsdatensatz benutzt, der ein komplexes Szenario darstellte aus ungefähr 100 Basisstationen, irregulärer Zellenverteilung und fluktuierendem Netzverkehr basierend auf Konzentrationspunkten. Die Datei für diesen Datensatz umfasste ungefähr 18 MB. Um den Messungsaufwand zu reduzieren, wurde der Rechenaufwand für eine Simulation auf 300 simulierte Sekunden reduziert. Diese benötigten auf den eingesetzten Rechenressourcen in etwa 20 Minuten echte Rechenzeit. In einer realistischen Untersuchung würde man ungefähr 3000 simulierte Sekunden benötigen. Die in diesem Experiment erzielten Ergebnisse sind dementsprechend konservativ.

In den zwei Graphen in Abbildung 10.2 sind die Messergebnisse als Beschleunigungs- und Effizienzwerte aufgetragen. Die Beschleunigung S_n bei Einsatz von n Rechnern ergibt sich dabei durch

$$S_n = \frac{t_1}{t_n},$$

wobei t_i die Laufzeit der Applikation bei Einsatz von i Rechnern ist. Für t_1 wurde ein dedizierter Rechner für die Simulationen benutzt, ohne dass die Infrastruktur zum Einsatz kam. Die absolute Effizienz E_n^{abs} bestimmt sich durch

$$E_n^{abs} = \frac{S_n}{n}$$

und stellt ein Maß dar für die Ausnutzung der zur Verfügung stehenden Ressourcen.

[4] Es wurden Sparc-Ultra-1 Rechner der Firma Sun eingesetzt.

[5] Fünf dieser Rechner standen in der Friedrich-Schiller Universität von Jena, die anderen in der Friedrich-Alexander Universität von Erlangen/Nürnberg.

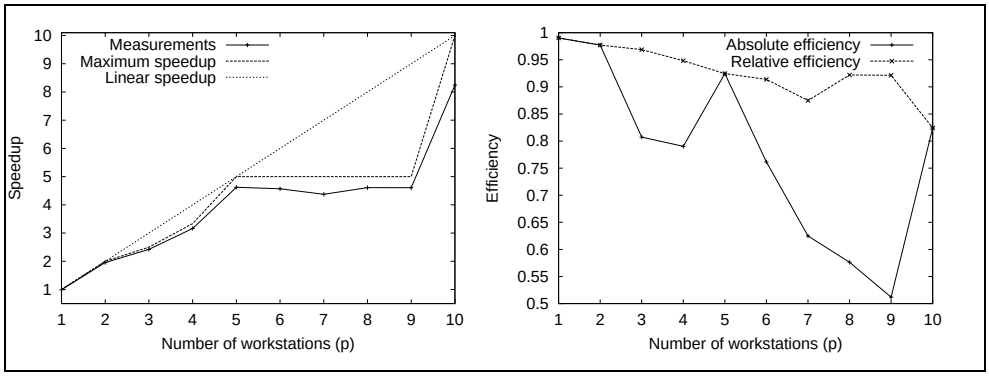

Abbildung 10.2.: Beschleunigung (Speedup) und Effizienz

Da die hier betrachtete Applikation nur eine geringe Nebenläufigkeit besitzt, wird die maximal mögliche Ausnutzung signifikant eingeschränkt, wie sich in dem linken Graphen erkennen läßt. Da das Ziel dieser Untersuchung die Betrachtung des durch den Einsatz von Amica zusätzlich betriebenen Aufwands war, wurde als zusätzliche Größe die *relative Effizienz* E_n^{rel} eingeführt. Diese setzt die erzielte Beschleunigung mit der durch die Applikation vorgegebenen maximalen Beschleunigung in Bezug:

$$E_n^{rel} = \frac{S_n}{S_{max}}$$

Trotz des hohen Kommunikationsaufwands, der durch die umfangreichen Umgebungsdaten erzeugt wurde, und des einfachen Verteilungsalgorithmus wurde eine ausreichende Effizienz von mehr als 80% bei dem Einsatz von 10 Rechnern erzielt. Wie an der relativen Effizienz zu erkennen ist, betrag der Verlust durch den Einsatz von Amica i.a. weniger als 10%. Ein Einsatz unter realistischeren Bedingungen mit höheren Simulationszeiten würde die Ergebnisse entsprechend verbessern.

Aufgrund der offenen und dynamischen Architektur können freie Ressourcen, die von der Applikation nicht benutzt werden, von anderen Applikationen genutzt werden. Neue Ressourcen, die in die Infrastruktur integriert werden, werden sofort von allen laufenden Applikationen genutzt. Dieses manuell oder mit Skripten zu gewährleisten und dabei auch die Eingabedaten zu verteilen und die Ausgabedaten einzusammeln, ist mühsam und aufwendig. Zusammengefasst kann man sagen, dass der Einsatz der Infrastruktur bei einem vertretbaren zusätzlichen Aufwand die Erstellung und den Ablauf verteilter grobkörniger Applikationen deutlich erleichtert.

10.2. Einbettung von MPI Komponenten

Wie in Abschnitt 7 diskutiert wurde, besteht ein Unterschied zwischen Applikationen für lokal verteilte Systeme wie Workstation-Cluster oder Hochleistungsparallelrechnern und Applikationen für stark verteilte System wie Me-

tacomputer oder das Grid. Der Ansatz von Amica, beide Ansätze zu kombinieren, besteht darin, dass Applikationen aus einzelnen Komponenten, bzw. Subapplikationen, bestehen, die für lokal verteilte Systeme konzipiert und optimiert sind. Die Applikation selber koordiniert diese Subapplikationen, indem sie ihren Ablauf synchronisiert und ihnen die zur Laufzeit zur Verfügung stehenden Ressourcen zuordnet.

Solche Subapplikationen setzen im allgemeinen auf Bibliotheken auf, die Kommunikations- und Synchronisationsprimitive anbieten, die optimiert für konkrete Systeme sind. Das hierfür wohl am häufigsten eingesetzte System ist das Message Passing Interface (MPI) [MF94]. In diesem Abschnitt wird beschrieben, wie Amica um einen zusätzlichen generischen Dienst erweitert wurde, der es erlaubt, Subapplikationen, die auf MPI aufsetzen, auf entfernten Rechenressourcen auszuführen. Der Einsatz von MPI ist exemplarisch. Andere Bibliotheken, wie z.B. PVM [GBD+94], lassen sich ähnlich integrieren.

Die Integration von MPI besteht aus der Einführung eines neuen Rechendienstes zur Ausführung der Subapplikation und eines neuen Datenobjektyps, um diese zu speichern. Eine Subapplikation wird von dem Anwendungsentwickler als Quellcode geliefert. Um diese auf entfernten Rechenressourcen auszuführen, werden sie dort zuerst konfiguriert und dann übersetzt, falls auf kein ausführbares Programm für den benötigten Ressourcentyp zurückgegriffen werden kann. Da Konfiguration und Übersetzung sehr von der Applikation abhängen, werden hierfür Skripte benutzt, die der Applikationsentwickler zur Verfügung stellen muss. Dabei kann er allerdings Standards wie Makefiles o.ä. einsetzen. Die Eingabedaten, die für die Ausführung benötigt werden, werden zu dem entfernten System übertragen. Die Ausgabedaten, die während und nach der Ausführung anfallen, werden abgegriffen und in Datenobjekten der Infrastruktur abgelegt.

Der neue Datenobjekttyp zur Speicherung des Quellcodes und ausführbarer Dateien von Subapplikationen heißt `Executable`. Seine CORBA Schnittstelle ist Abbildung 10.3 zu entnehmen. Es wird mit dem Typ `BinaryType` zwischen zwei Arten von ausführbaren Dateien unterschieden: einer einzelne Datei und einem Archiv von Dateien mit einer Hauptdatei, deren Name in dem Attribut `mainProg` gespeichert ist. Ausführbare Dateien benötigen evtl. eine Laufzeitunterstützung. Hier wird mit dem Aufzählungstyp `RuntimeSupport` nur „keine Unterstützung“ und MPI angeboten. Die Struktur `Executable` dient nun zum Speichern ausführbarer Dateien. Ihr Attribut `binary` enthält in Abhängigkeit des Wertes von `binaryType` entweder eine ausführbare Datei oder ein mit `gzip` komprimiertes `Tar`-Archiv.

Quellcode wird in der Struktur `SourceCode` gespeichert. Diese besteht hauptsächlich aus einem komprimierten Archiv, das in dem Attribut `archive` gespeichert ist und einigen Befehlen, bzw. Namen von Skriptdateien, die für die Konfiguration und Übersetzung notwendig sind. Außerdem enthält sie die Spezifikation der geforderten Laufzeitunterstützung und eine Beschreibung der Art der ausführbaren Datei, die nach der Übersetzung in der in `execFileName` beschriebenen Datei zu finden ist.

Die eigentliche Schnittstelle des Datenobjekttyps besteht aus dem Zugriff auf den Quellcode in dem Attribut `sourceCode` und den ausführbaren Datei-

```
interface Executable : DataObject
{
  /* type of the executable binary */
  union BinaryType switch (char)
  { case 's':
       boolean single;
    case 'a':
      struct  BinaryArchive { string mainProg;  } archive ;
  };
  enum RuntimeSupport {NONE,MPI};

  struct Executable
  { BinaryType binaryType;
    OctetSeq   binary;
    RuntimeSupport runtimeSupport; };

  struct SourceCode
  { string     compileCommand;  /* build executable*/
    string     configureCommand;/* configure build process*/
    string     execFileName;    /* name of the executable */
    BinaryType execType;        /* type of the executable */
    RuntimeSupport execRuntSupp;
    OctetSeq   archive; };

  attribute SourceCode sourceCode;

  /* returns true iff a binary exists
     for a given architecture. */
  boolean existsForArchitecture(
      in computation::CUArchitecture arch);

  /* adds a new executable. */
  void addExecutable(in computation::CUArchitecture arch,
                     in Executable prog);

  /* returns an executable for a given architecture. */
  Executable getExecutable(
      in computation::CUArchitecture arch);
}
```

Abbildung 10.3.: Die CORBA Schnittstelle für den Datenobjekttyp `Executable`

en. Diese werden über den Architekturtyp der Rechner, auf denen sie lauffähig sind, abgelegt. Um dem Applikationsentwickler und dem Benutzer den Umgang mit diesen Datenobjekten zu erleichtern, stehen Adapterkomponenten zur Verfügung, die ein Initialisieren über statische Eigenschaften oder mit einer grafischen Schnittstelle zur Laufzeit erlauben. Abbildung 10.4 zeigt zwei Bildschirmfotos mit Dialogboxen für den Zugriff auf diese Datenobjekte.

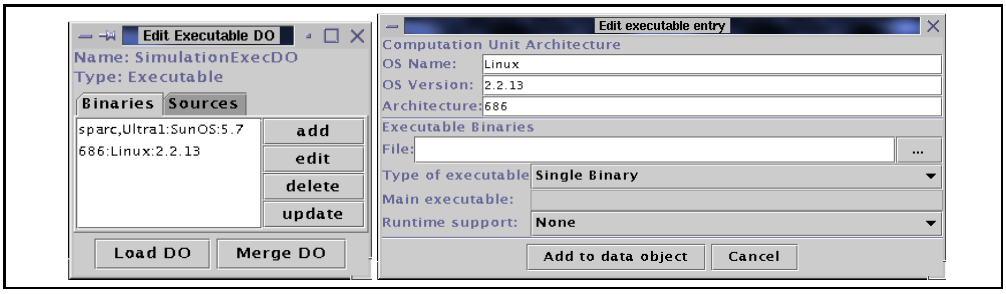

Abbildung 10.4.: Interaktiver Zugriff auf `Executable` Datenobjekte mit Adapterkomponenten zur Laufzeit

Um nun eine Subapplikation, die in einem Datenobjekt abgelegt wurde, zu starten, wurde der neue Rechendienst `execution` eingeführt. Eine Dienstkomponente, die diesen Dienst nutzt, wird mit dem `Executable` Datenobjekt und den Datenobjekten für die Ein- und Ausgabe verbunden. Desweiteren ist sie mit einer Mindest- und Maximalanzahl von Prozessoren parametrisierbar, die für die Ausführung benötigt werden.

Bei Aktivierung der Dienstkomponente wird zuerst der geeigneteste Dienstanbieter gesucht, der die gestellten Anforderungen erfüllt. Als Kostenmaß wird die Auslastung der zur Verfügung stehenden Rechner und die Länge der Warteschlangen eingesetzt. Wird nun der erzeugte Dienst gestartet, muss überprüft werden, ob ausführbarer Code für die bestimmte Rechnerarchitektur vorliegt. Falls nicht, muss dieser generiert werden. Abbildung 10.5 zeigt ein Arbeitsablaufdiagramm des Prozesses, welcher hierfür durchgeführt wird.

Existiert ausführbarer Code in Form einer Datei wird dieser in das lokale Dateisystem kopiert. Ist es statt dessen ein Archiv, wird es kopiert und entpackt. Danach werden die Eingabedaten aus den Datenobjekten in das lokale Dateisystem kopiert und das Programm gestartet. Nach dessen Terminierung werden die Ergebnisdaten in Datenobjekte übertragen. Hinzukommend ist es möglich, den Inhalt eines Datenobjektes in die Standardeingabe der laufenden Subapplikation zu leiten und andererseits dessen Standardausgabe in ein Datenobjekt.

Existiert dagegen kein ausführbarer Code bei vorhandenem Quellcode, wird solcher nach einer Konfigurations- und Übersetzungsphase generiert. Das Ergebnis wird dann in dem Datenobjekt abgelegt. Sollte der Prozess scheitern, also den Zustand `failure` erreichen, wird automatisch versucht eine andere Rechenressource zu nutzen.

Bis jetzt wurde noch nicht beschrieben, wie der Anwendungsentwickler den Bruch zwischen dem Kommandozeilen-orientierten Start eines

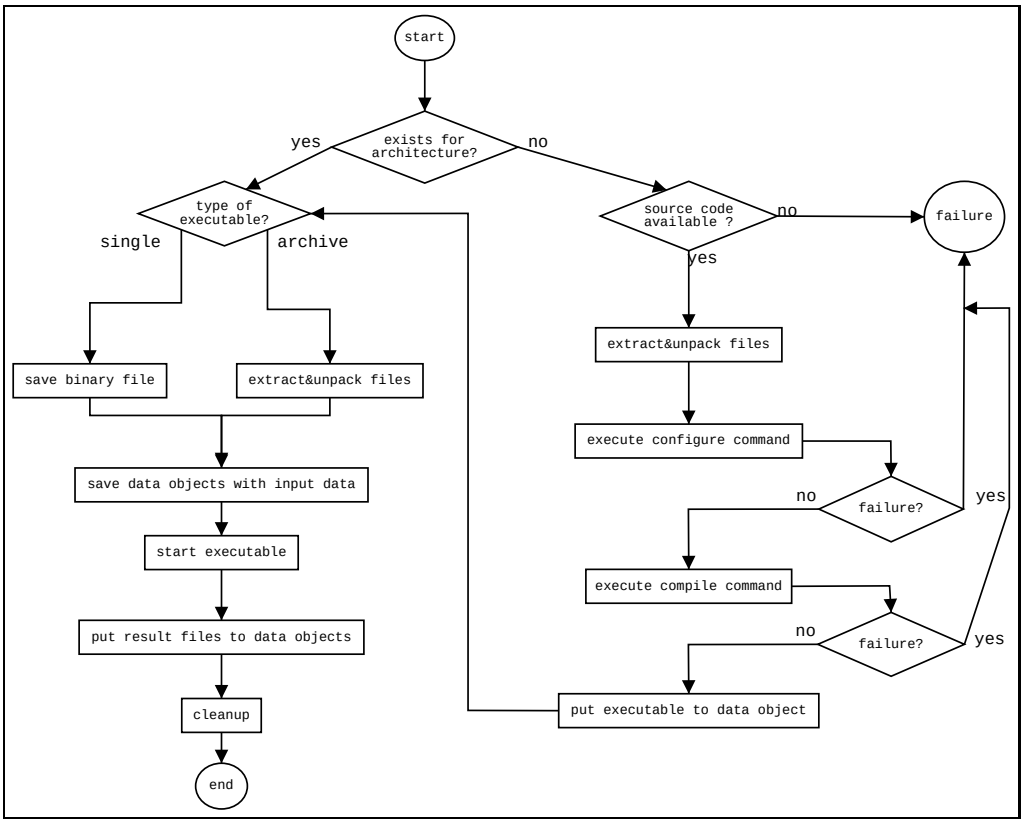

Abbildung 10.5.: Arbeitsablaufdiagramm für die Ausführung einer Subapplikation

Programms und dessen Einbettung in die Applikationsarchitektur als `RemoteService`-Komponente überbrückt. Dies wird über ein Komponentenattribut `Parameters` erreicht, das eine Zeichenkette mit den dafür notwendigen Informationen enthält. Sie besteht hauptsächlich aus den Kommandozeilenparametern, welche dem ausführbaren Programm beim Start übergeben werden. Zusätzlich enthält sie Fragmente, welche die Verbindung zu den Datenobjekten beschreiben. Diese Fragmente werden in drei Phasen bearbeitet:

1. Alle spezifizierten Datenobjekte, die als Eingabe benutzt werden, werden in das lokale Dateisystem geladen.

2. Die Fragmente werden geeignet substituiert (s.u.). Die entstehende Zeichenkette wird für den Start der Subapplikation benutzt.

3. Nach der Terminierung werden die Ergebnisse in die spezifizierten Datenobjekte übertragen.

Ein allgemeines Entwurfsziel bei der Spezifikation besteht darin, dass alle Abhängigkeiten zwischen den Architekturelementen in der Architektur ausgedrückt werden. Deswegen dürfen die Fragmente sich nicht direkt auf Datenobjekte beziehen, sondern nur auf die Anschlüsse der Komponente, die dann über einen Konnektor mit einem Datenobjekt verbunden sind. Als Kurzschreibweise wird im folgenden Abschnitt über die einzelnen Fragmenttypen die Notation *dob(port)* benutzt, um das Datenobjekt zu bezeichnen, das mit dem Anschluss *port* verbunden ist.

- `$<(#port,file)`
  Der Inhalt des Datenobjekts *dob(port)* wird in der ersten Phase in eine Datei mit dem Namen `file` geschrieben. Das Fragment wird durch den Pfad konkateniert mit `file` substituiert.

- `$>(#port,file)`
  Der Inhalt der Datei `file` wird in der dritten Phase in das Datenobjekt *dob(port)* geschrieben. Das Fragment wird durch den Pfad konkateniert mit `file` substituiert.

- `-<#port`
  Der Inhalt des Datenobjekte *dob(port)* wird in der zweiten Phase in die Standardeingabe der ausführbaren Datei geleitet. Das Fragment wird aus der Kommandozeile entfernt.

- `->#port`
  Die Daten aus der Standardausgabe, die in der zweiten Phase entstehen, werden in das Datenobjekt *dob(port)* geleitet. Das Fragment wird entfernt.

- `-2>#port`
  Die Daten aus der Fehlerausgabe werden in das Datenobjekt *dob(port)* geleitet. Das Fragment wird entfernt.

- `$$#port`
  Das Fragment wird in der zweiten Phase durch den Inhalt des Datenobjekts *dob(port)* ersetzt.

Dieser Dienst `execution` zur Ausführung von Subapplikationen wurde in Java für einzelne Multiprozessorrechner mit gemeinsamen Speicher und für Workstation-Cluster implementiert. Für letztere wurde eine Infrastruktur realisiert, in der die einzelnen Rechner von Hintergrundprozessen verwaltet werden, die einerseits die Auslastung periodisch ermitteln, andererseits die Ausführung von Programmen ermöglichen. Die Amica-Recheneinheit verwaltet alle Hintergrundprozesse eines Workstation-Clusters und stellt ihn auf diese Weise der Amica-Infrastruktur als eine Rechenressource zur Verfügung. Als konkrete MPI-Bibliothek wurde MPICH eingesetzt [mpi02].

Das System wurde unter anderem mit einer Applikation getestet, die als Subapplikation eine parallele Strömungsberechnung einsetzt. Abbildung 10.6 zeigt die zugehörige Applikationsarchitektur. Nach einer Initialisierung des Datenobjekts `FluidComputation` mit dem Quellcode der Strömungsberechnung und des Datenobjekts `parameter` mit einigen Parametern für die Berechnung, wird die Subapplikation ausgeführt. Die Ergebnisse werden in dem Datenobjekt `ResultVectorField` abgelegt und abschließend mit einer Adapterkomponente visualisiert, die Strömungslinien berechnet und als Grafik anzeigt.

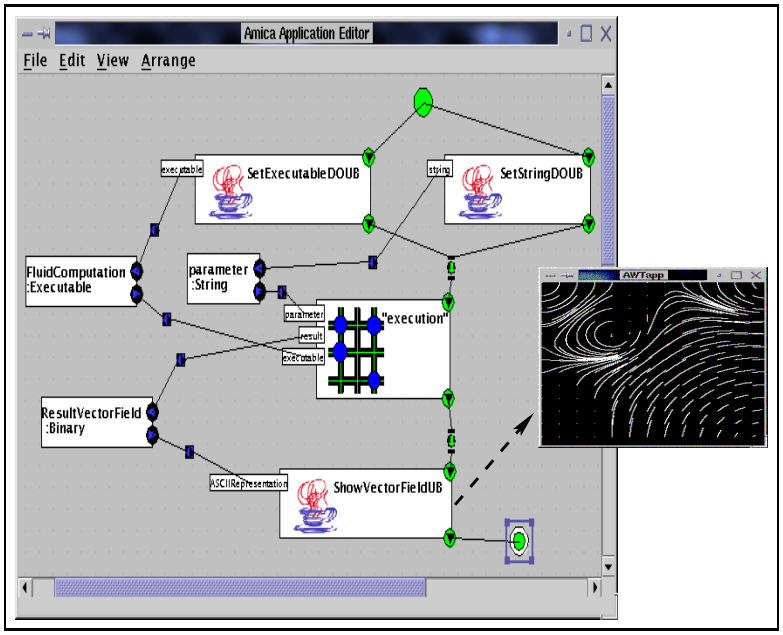

Abbildung 10.6.: Strömungsberechnung als Beispiel für die Einbettung einer MPI-Subapplikation

Ein Vorteil dieses Ansatzes, der sich bei dieser Applikation darstellt, besteht in der Komfortsteigerung für den Applikationsentwickler und den Anwender. Das manuelle Suchen einer Ressource, die Übersetzung und Konfiguration der

Subapplikation und die Übertragung der Daten wird ihm abgenommen. Um den durch den Einsatz von Amica erzeugten zusätzlichen Aufwand zu bestimmen, wurde eine einfache Applikation implementiert, die ein kleines Programm in ein Datenobjekt schreibt und dann ausführt. Bei der Ausführung auf einem Rechner[6] betrug die Zeit zwischen dem Start der übersetzten Applikation und dem Start des eigentlichen Programms 529 msec. Dieses ist schon bei Rechenzeiten ab einer Minute absolut vertretbar.

10.3. Paradigmenbruch durch Kombination von Strömungskonnektoren mit Subapplikationen

Wie im vorherigen Abschnitt deutlich wurde, besteht eine Hauptanwendung des Amica-Systems in der Koordination von Subapplikationen. Solche Subapplikationen sind aber im allgemeinen nach dem Batch-Prinzip konzipiert, d.h. sie werden mit Eingabedaten versorgt, führen ihre Berechnungen durch, schreiben ihre Ergebnisse und terminieren. Dieses Verhalten harmoniert gut mit dem reinen Einsatz von Datenobjekten, fügt man allerdings Datenströme zu der Applikationsarchitektur hinzu, ergibt sich ein Paradigmenbruch, da die Subapplikationen i.a. nicht für eine Stromverarbeitung vorgesehen sind. Der eingesetzte Rechendienst `execution` (s. Abschnitt 10.2) muss daher versuchen, diesen Bruch zu überbrücken.

Die Anbindung von Datenobjekten an diesen Ausführungsdienst über Fragmentsubstitution in der Kommandozeile wurde im vorherigen Abschnitt 10.2 beschrieben. Ist nun ein Datenstrom mit einem Anschluss einer Rechenkomponente zur Ausführung einer Subapplikation verbunden, so wird jedes eingehende Stromelement wie ein eigenes Datenobjekt behandelt. Jedes von der Ausführung erzeugte Ergebnis wird als ein Stromelement in den verbundenen ausgehenden Datenstrom geleitet. Die Syntax der Fragmente braucht auf diese Weise nicht verändert zu werden.

Ist eine solche Rechenkomponente nur mit Datenobjekten verbunden, kann sie nur explizit über ein eingehendes Aktivierungssignal aus einem Kontrollflusskonnektor vom Typ `OperationCn` aktiviert werden (s. Abschnitt 6.1). In einem stromorientierten Paradigma sind Komponenten aber stets aktiv und warten auf eingehende Stromdaten. Wie in Abschnitt 9 schon beschrieben wurde, sind die in Amica benutzten Rechenkomponenten daher so erweitert wurde, dass eine Aktivierung sowohl explizit als auch durch eingehende Stromdaten erfolgt.

Die Dienstkomponente für die Ausführung von Subapplikationen wartet nun bei Aktivierung darauf, dass von allen mit ihr verbundenen eingehenden Datenströmen mindestens ein Datenelement vorliegt. Ist dies der Fall, wird die Subapplikation, wie im vorherigen Abschnitt beschrieben, ausgeführt. Wenn mindestens ein eingehender Datenstrom terminiert, beendet sich die Dienstkomponente und, sollte sie mit einem Kontrollflusskonnektor verbunden sein, sendet sie ein Aktivierungssignal an diesen[7].

[6] Standard PC mit einem Pentium III Prozessor mit 450 MHz

[7] Die noch ausstehenden Stromelemente der anderen Datenströme bleiben bestehen und wer-

Die in Abbildung 10.7 dargestellte Abbildung zeigt exemplarisch eine verteilte Applikation als Kombination von Arbeitsablaufkomponenten, also Rechenkomponenten nach dem Batch-Konzept, und Strömungskomponenten. Dabei wird ein Film berechnet, indem zuerst die einzelnen Bilder aus dreidimensionalen Szenarien erzeugt werden, die dann zu einem Film zusammengefügt werden. Dabei kommen nur frei erhältliche Werkzeuge zum Einsatz, die als Subapplikationen zusammengesetzt werden.

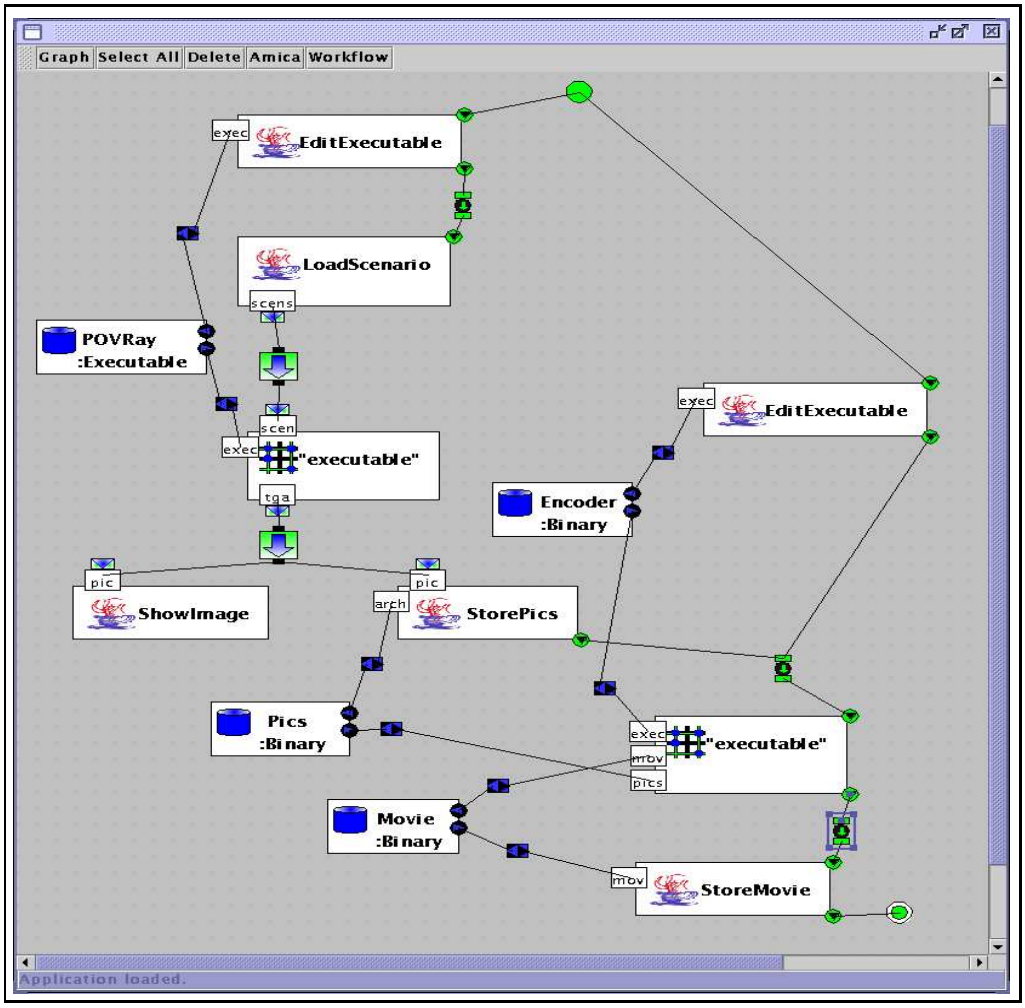

Abbildung 10.7.: Verteilte Berechnung eines computer-generierten Films als Beispiel für den Einsatz von Datenströmen

Für die Bilderzeugung wird der Raytracer POV-Ray [pov02] eingesetzt, für die Videokodierung das Programm `mpeg`. Diese werden als erster Schritt der Applikation mit parametrisierten Standardadapterkomponenten

den bei der nächsten Ausführung der Subapplikation eingesetzt. Dafür muss sich der terminierte Datenstrom neu etablieren. Ein alternativer Ansatz wäre es, darauf zu bestehen, dass alle eingehenden Datenströme terminieren, damit die Dienstkomponente sich beendet. Diese Strategie wurde aber als zu restriktiv bei keinem erkennbaren Vorteil verworfen.

`EditExecutableDOGUIUB` als ausführbare Subapplikationen in den entsprechenden Datenobjekten gespeichert. Dann werden aus dem lokalen Dateisystem die dreidimensionalen Bildszenarien geladen und in einen Datenstrom gespeist.

Dieser Datenstrom fließt in eine Dienstkomponente, welche den Bilderzeuger mit dem Szenario ausführt und das erzeugte Bild in einen weiteren Datenstrom leitet. Der zugehörige Stromkonnektor benutzt für die Erzeugung des ausgehenden Datenstroms die Rundsendungspolitik, d.h., die beiden angeschlossenen Rechenkomponenten erhalten die gleichen Bilddaten. Die linke Adapterkomponente visualisiert den aktuellen Berechnungsstatus, indem es das aktuelle Bild in einem Fenster anzeigt. Die rechte Adapterkomponente speichert die erzeugten Bilder als Archiv in dem Datenobjekt `Pics`.

Abschließend werden die Bilder mit einer Dienstkomponente zu einem Film zusammengefügt und dieser dann von einer Adapterkomponente im lokalen Dateisystem gespeichert. Hier stellt sich die Frage, warum die Bilder nicht als Datenstrom in den Videokodierer geleitet werden. Dies würde die Infrastruktur entlasten, da dann eine Zwischenspeicherung der Bilder in einem Datenobjekt nicht mehr nötig wären. Zusätzlich wären Effizienzsteigerungen durch den erhöhten Grad an Nebenläufigkeit möglich. Es gibt drei Ursachen hierfür:

- Die hier vorgestellte Einbettung von Subapplikationen unterstützt kein Konzept eines inneren Zustands der Rechenkomponente. Für jeden Satz an Eingabedaten wird die Subapplikation neu initialisiert. Ein Videokodierer muss sich aber die bisher gesehenen Bilder merken, um gute Kompressionsraten zu erzielen. Ein allgemeiner Lösungsansatz für dieses Problem besteht in einer expliziten Speicherung des Zustands in einem Datenobjekt, welches der Subapplikation zur Verfügung steht.

- Dieser Gebrauch von Datenströmen beschreibt eine reduzierende Filteroperation. Aus n eingehenden Datenelementen werden m Ergebniselemente erzeugt mit $m < n$. Die Einbettung von Subapplikationen geht aber davon aus, dass $n = m$ gilt. In diesem Fall ist $m = 1$ und die Dienstkomponente könnte das Ergebnis in einem Datenobjekt ablegen. Die Erzeugung eines ausgehenden Datenstroms mit $n \neq m$ ist aber prinzipiell nicht möglich. Dieses Problem hängt konzeptionell zusammen mit dem obigen Problem des fehlenden Zustands, ergibt sich aber konkret direkt aus dem Mechanismus der Einbettung.

- Die benutzte Subapplikation unterstützt das Strömungsparadigma nicht, so dass es ohne eine Anpassung nicht möglich ist, sie als vollwertige Strömungskomponente einzusetzen. Hier zeigen sich prinzipielle Einschränkungen bei der Komposition von Komponenten aus Altsoftware, die einer Paradigmenheterogenität unterliegen. Stünde eine Subapplikation zur Verfügung, die das Strömungsparadigma unterstützt, könnte man diese wie in Abschnitt 10.1 gezeigt, leicht als neuen speziellen Dienst in die Infrastruktur integrieren.

Als Fazit lässt sich feststellen, dass die hier vorgestelle Einbettung eine Benutzung von Subapplikationen als Strömungskomponenten sinnvoll und vor-

teilhaft ermöglicht. Allerdings ist diese Einbettung prinzipiellen Beschränkungen unterworfen, die sich aus der Paradigmenheterogenität zwangsläufig ergeben.

10.4. Diskussion und offene Punkte

Die Fallstudien für das Amica-System haben gezeigt, dass es sich für die Implementierung von rechenintensiven Applikationen mit grobkörniger Nebenläufigkeit gut eignet. Das in den verschiedenen Erweiterungsmodulen bereit gestellte Vokabular ermöglicht eine komfortable und adäquate Erstellung der Applikationen. Die einzelnen Architekturelemente sind leicht verständlich und benötigen nur eine geringe Parametrisierung. Interaktive Funktionalität oder kleinere lokale Berechnungen sind allerdings von dem Applikationsentwickler in Form von Java-Klassen selbst zu implementieren.

Die Infrastruktur und die entsprechenden Architekturelementtypen unterstützen aufgrund ihrer generischen Architektur gut die Integration von Alt-Software. Es zeigten sich allerdings leichte Probleme bei dem Einbetten von Alt-Software in Architekturelemente, die unterschiedliche Koordinationsparadigmen einsetzen. In der im vorherigen Kapitel 10.3 beschriebenen Fallstudie wurde eine ad hoc-Lösung vorgestellt. Hier wäre es vorteilhaft, eine allgemeine Lösung zu finden, wie sich unterschiedliche Koordinationsparadigmen miteinander vereinbaren lassen. Dieses ist allerdings Gegenstand aktueller Forschungen, siehe z.B. [Löh02, Löh03].

Der zusätzliche Kommunikations- und Rechenaufwand aufgrund des Einsatzes des ECL-Systems und der Amica-Infrastruktur ist vernachlässigbar, wenn die einzelnen Rechenkomponenten eine Rechenzeit von mehreren Minuten oder mehr benötigen. Dieses ist in der Applikationsklasse, auf die Amica abzielt, gegeben.

Zwei für die Applikationsklasse wichtige Eigenschaften besitzt das Amica-System allerdings noch nicht in ausreichender Qualität. Diese sind Fehlertoleranz und Unterstützung von Abrechnungsmodellen. Die Forderung nach Fehlertoleranz ist deshalb so relevant, weil zusätzlich zu der allgemeinen erhöhten Fehleranfälligkeit verteilter Systeme noch hinzukommt, dass sich die in Amica benutzten Ressourcen in unterschiedlichen Verwaltungsbereichen befinden. Es ist daher z.B. jederzeit möglich, dass entfernte Rechner ohne vorherige Warnung zu Wartungszwecken einfach ausgeschaltet werden. Eine laufende Applikation sollte derartige Ressourcenausfälle tolerieren können.

Es gibt verschiedene Ansätze, mit welchen Erweiterungen des Amica-Systems sich diese Anforderung erfüllen läßt. Eine verbreitete Methode besteht darin, den Zustand einer Applikation in regelmäßigen Abständen zu speichern (Checkpointing). Neben dem Zustand aller Applikationsobjekte, müsste dann auch der Zustand der Datenobjekte, der evtl. sehr umfangreich ist, gespeichert werden und es müsste zusätzlich sicher gestellt werden, dass keine Nachrichten verloren gehen, die sich gerade im Netzwerk befinden. Abschließend ist sicherzustellen, dass die gespeicherten Zustände der einzelnen Objekte nicht kausal voneinander abhängig sind. Auf die Begründung

der letzten Forderung wird hier verzichtet und statt dessen auf [NX95] verwiesen. Derartige kausalen Abhängigkeiten entstehen zwischen den einzelnen Replikaten der Datenobjekte, zwischen Rechendiensten, Datenobjekten und Strömungskanälen und zwischen Rechendiensten, Dienstfabriken und Recheneinheiten. Damit ist dieser Ansatz zwar möglich, aber auch aufwendig.

Eine Variation dieses Ansatzes besteht darin, die einzelnen Operationen in dem Amica-System als Transaktionen anzusehen, die beim Auftreten eines Fehlers wieder rückgängig gemacht werden können (Rollback). Hier bietet es sich an, die Ausführung der einzelnen Rechenoperationen als Transaktionen anzusehen. Vor der Ausführung wird der Applikationszustand gespeichert und bei einem Fehler wieder eingespielt. Da hier die möglichen Zeitpunkte der Speicherung eingeschränkt sind, lässt sich diese Speicherung evtl. einfacher durchführen als in dem vorherigen Ansatz. Probleme ergeben sich allerdings, wenn zwei Rechenoperationen gleichzeitig auf die gleichen Datenobjekte zugreifen. Diese lassen sich aber mit Techniken lösen, die für Datenbanksysteme entwickelt wurden. Schwieriger ist die Speicherung des Systemzustands vor der lokalen Ausführung von Java-Code. Hier muss eigentlich der gesamte Zustand des lokalen Systems gespeichert werden. Zusätzlich ist es auch unklar, wie Rechenoperationen, die auf Stromdaten arbeiten, zu behandeln sind.

Insgesamt ist auch dieser Ansatz aufwendig. Er wird aktuell in sehr vereinfachter Form unterstützt, indem bei einem Fehler während der Ausführung eines entfernten Rechendienstes nochmals versucht wird, einen neuen entfernten Rechendienst zu erzeugen und auszuführen. Hat der fehlerhafte Rechendienst allerdings vorher auf Datenobjekte zugegriffen, kann das Ergebnis des Ersatzrechendienstes evtl. fehlerhaft sein.

Ein Alternative besteht darin, dass weder die Infrastruktur noch die Applikationsobjekte oder die Laufzeitunterstützungen eine Fehlertoleranz anbieten, sondern die Fehlerbehandlung dem Applikationsentwickler überlassen. Bei diesem Ansatz ist das Vokabular so zu erweitern, dass ein Kontrollfluss für die Fehlerbehandlung unterstützt wird. Allerdings bricht dieser Ansatz mit der Strategie, dass der Applikationsentwickler sich möglichst wenig mit der Infrastruktur auseinanderzusetzen hat.

Der zweite offene Punkt in dem Amica-System ist die Unterstützung von Abrechnungsmodellen. Aktuell werden entfernte Rechenressourcen für akademische Projekte i.a. kostenfrei zur Verfügung gestellt. Es ist aber davon auszugehen, dass sich, nachdem sich eine allgemeine Infrastruktur für die Ausführung rechenintensiver Applikationen in verteilten Systemen etabliert hat, Geschäftsmodelle entwickeln, um eigene Rechenressourcen interessierten Anwendern kostenpflichtig zur Verfügung zu stellen. Im Rahmen einer Diplomarbeit [Bal00] wurde das Amica-System um Zugriffsschutz und um Protokollierung der Nutzung von Recheneinheiten erweitert, so dass die für die Unterstützung solcher Geschäftsmodelle notwendigen technischen Grundlagen vorhanden sind.

Für ein vollständiges Abrechnungsmodell sind aber noch zusätzliche Erweiterungen notwendig. Insbesondere muss der Anwender seiner Applikation ein Budget zur Verfügung stellen, welches für deren Ausführung zur

Verfügung steht. Zur Laufzeit muss die Applikation dann nach vorgegebenen Strategien versuchen, die benötigten Rechenressourcen einzukaufen. Neben implementierungstechnischen Änderungen fehlt für die Integration in Amica hauptsächlich noch ein mathematisches Modell, das die Applikation und die Infrastruktur einsetzen kann, um über Preis und Akzeptanz eines Preises zu entscheiden. Dieses ist Gegenstand aktueller Forschung [Buy02, MTH02, CCG02].