Das
D Framework

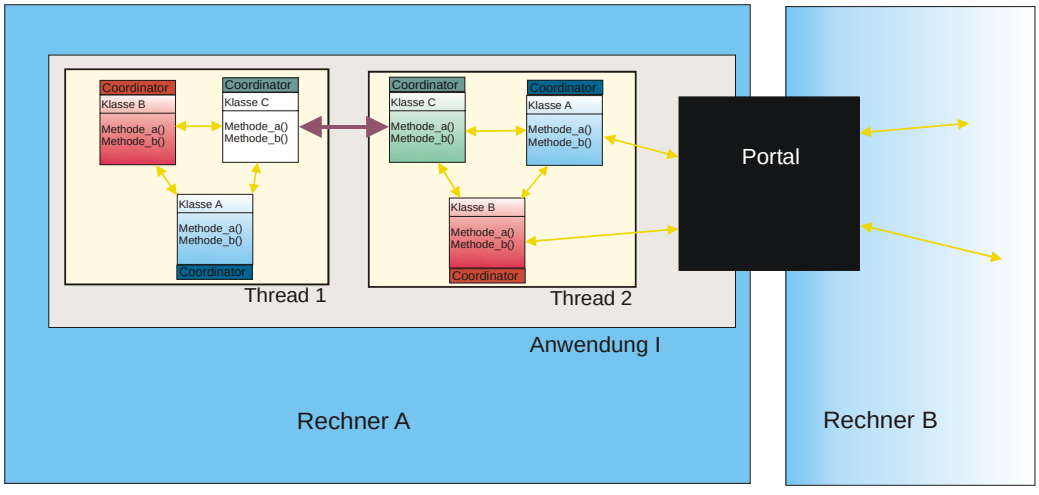
1. Einleitung & Motivation

1.1. Das Problem

Verteilte Software zu entwickeln ist keine leichte Aufgabe. Da das Programm sich über mehrere Threads oder Prozesse, ja vielleicht sogar über Rechnergrenzen hinweg verteilt, müssen diese Programmteile alle miteinander kommunizieren können. Das bedarf aber einer aufwändigen Synchronisierung und die Einbindung von Fernaufrufen von Prozeduren. Der Entwickler muss sich also schon zur Entwurfszeit Gedanken über die Sicherung seiner Klassen und deren Synchronisierung machen. Das ist aufwändig und hat mit dem eigentlichen Problem nichts zu tun. Es erfordert außerdem Wissen des Programmierers über verteilte Systeme, was vielleicht bei Teamwork in größeren Projekten nicht immer der Fall ist. Es wäre schön, wenn derjenige, der sich mit dem zu lösenden algorithmischen Problem befasst, sich voll und ganz auf die Bearbeitung beschränken kann und später in einem weiteren Arbeitsschritt die Verteilung auf mehrere Prozesse bzw. Rechner vorgenommen wird. Das soll natürlich klar voneinander getrennt sein, so dass der Code für die Verteilung nicht im Programmcode des eigentlichen Problems auftaucht. Das wäre also ein erstklassiges Einsatzgebiet für aspektorientierte Programmierung.

1.2. D Framework

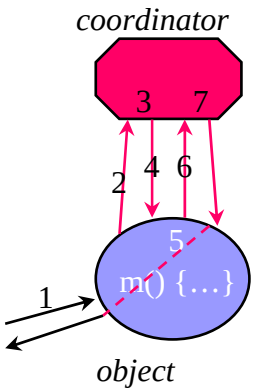
Das D Framework ist eine Komposition aus mehren Programmiersprachen. Es soll dem Programmierer die Möglichkeit geben, verteilte Software zu entwickeln, die mit Hilfe von Aspekten einfacher zu schreiben ist, da die Anwendung im Designprozess erst einmal unabhängig von den Synchronisations- und Kommunikationsproblemen geschrieben werden kann. Die querschneidenden Belange (*cross-cutting concerns*) werden dann erst mit Hilfe des D Frameworks implementiert und somit die Applikation für die verteilten Betrieb erweitert.
Das D Framework ist keine eigene Programmiersprache – deswegen auch der Name Framework. Es benutzt zwei Sprachen (RIDL und COOL) für die jeweils unterschiedlichen Belange bei verteilten Problemen. Die eigentliche Anwendung wird in einer dritten, vom D Framework unabhängigen Sprache geschrieben, der Komponentensprache. An diese Sprache gibt es nur geringe Anforderungen – so muss sie zum Beispiel objektorientiert sein, um das *information-hiding* Konzept vom D Framework zu unterstützen. Die Unterstützung von Threads ist natürlich auch sinnvoll, da sonst eine Verteilung nur schwer zu erreichen wäre. Es gibt bereits einige Ansätze, die Java als Komponentensprache benutzen (DJ).
Probleme werden in zwei verschiedene Bereiche eingeteilt. Die Sprache COOL übernimmt die Aufgabe, die Thread-Synchronisation bei der Ausführung der Komponenten zu überwachen und zu steuern. Mit RIDL wird die Kommunikation der einzelnen Programmkomponenten bei der Verteilung auf mehrere Rechner über Netzwerke hinweg gesteuert.
Im Folgenden werden die Sprachelemente von COOL und RIDL näher beleuchtet und dann anhand einiger Beispiele die Leistungsfähigkeit von D gezeigt.



2. COOL – Thread Synchronisierung

2.1. Allgemeines

COOL (*coordination language*) unterstützt die Verteilung der Programmstruktur auf Threads. Es bietet dafür Methoden zur Synchronisierung. Diese Methoden sind weitgehend unabhängig von den Klassen der Komponentensprache auch wenn sie diese schützen bzw. steuern.
Ein typisches COOL Programm besteht aus einer Reihe von so genannten *coordinator modules*. Diese *coordinator modules* oder kurz *coordinators* sind auf Basis von Klassennamen mit den einzelnen Klassen verbunden. Ein einzelner *coordinator* kann auch mehrere Klassen verwalten.
Coordinators sind eine Art Schutzmantel für Klassen. Sie kümmern sich um Anfragen für Methoden und sorgen für die korrekte Synchronisation ohne aber direkt in den Klassen implementiert zu sein. Die kleinste für *Coordinators* verwaltbare Einheit sind Methoden.
Coordinators sind keine Klassen und sie können auch nicht direkt instanziiert werden. Sie werden zusammen mit ihrer zugeordneten Klasse geladen und sind mit ihr dauerhaft verbunden. Die Kommunikation zwischen Klasse und *coordinator* läuft dabei nach einem fest vorgegebenen Schema ab.
Wie man leicht sieht, hängt die Implementierung des *coordinators* stark von der Struktur der Klasse ab. Der Code eines *coordinators* bezieht sich also immer auf eine bestimmte Klasse. Die Klassen selber haben aber einen Zugriff und Einfluss auf den *coordinator*, da der ja meist in der Komponentensprache gar nicht vorgesehen ist. Deswegen kann der *coordinator* auch nur auf für ihn sichtbare Elemente der Klasse zugreifen.



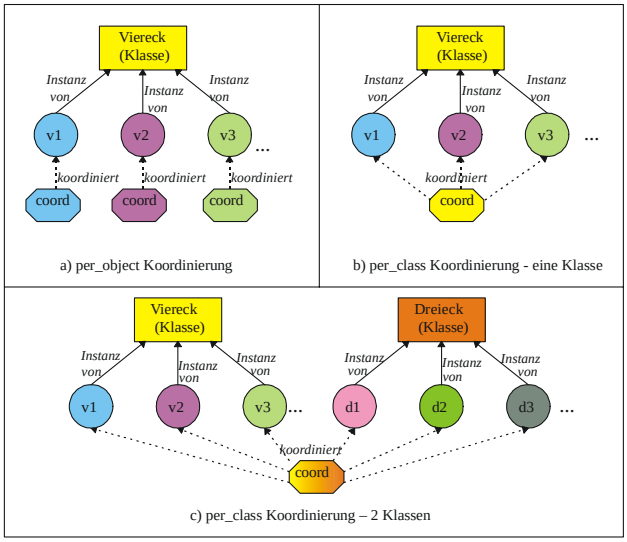
Quelle: [CL2]

2.2. Sichtbare Elemente der Klassen

COOL erkennt sämtliche Signaturen und Methoden einer Klasse, hat also auch Zugriff auf sämtliche *public*, *protected* und *private* Variablen.

2.3. Coordinator Deklaration

Wie schon vorhin erwähnt, kann ein *coordinator* auch mehrere Klassen verwalten. Es kann der *coordinator* so eingestellt werden, dass er entweder für jede Instanz der Klasse neu angelegt wird (*per_object coordination*), dass er alle Instanzen einer Klasse verwaltet (*per_class coordination*) oder dass er mehrere verschiedene Klassen (bzw. deren Instanzen) kontrolliert.



2.4. Coordinator Body

Der *coordinator body* definiert den Synchronisierungsstatus und die Steuerungsroutinen für die Sperrung bzw. Behandlung von parallelen Aufrufen von Methoden der zu steuernden Klasse. Die Steuerung arbeitet auf Basis der Methoden. Sie sind das kleinste zu steuernde Element.
Wenn ein Thread versucht, eine Methode aufzurufen, dann können zwei Dinge passieren: Wenn die Methode nicht im *coordinator body* benannt wurde, so wird sie unverzüglich ausgeführt.
Wurde sie benannt, wird geprüft, ob es Gründe gibt, dass der Aufruf verzögert wird. Hierfür können bestimmte Sperrmechanismen benutzt werden, die sicherstellen, dass die Klasse in jedem Fall einen konsistenten Zustand behält. Der Aufruf wird erst dann ausgeführt, wenn alle Bedingungen erfüllt sind. Solange muss der aufrufende Thread gegebenenfalls warten.

<CoordinatorBody>::=
{
[<CondVarDeclaration> {<CondVarDeclaration>}]
[<VariableDeclaration> {<VariableDeclaration>}]
[<SelfExclusiveMethods>]
[<MutuallyExclusiveMethodSet> {<MutuallyExclusiveMethodSet>}]
[<MethodManager> {<MethodManager>}]
}

2.5. Zustandsvariablen (c*ondition-variables*)

Condition-variables (kurz *conditions*) speichern den Zustand, in dem sich die zu überwachende Klasse befindet. Diese Zustände sind für die zu schützenden Methoden wichtig. Mit den *conditions* arbeiten die *Mutex* bzw *selfex* Funktionen um die Sperrungen zu realisieren. *Condition-variables* können nur TRUE oder FALSE annehmen. Die *condition-variable* Deklarationen geben den *conditions* einen Namen, mit dem sie innerhalb des *coordinators* benutzt werden können.
Syntax:

<CondVarDecl>::=
condition **<***VariableDeclarator> [{,* **<***VariableDeclarator>}] ;*

<VariableDeclarator>::=
<Identifier> **=** **<***CondVarInitializer>* |
*<Identifier***[]>** **=** *<CondArrayInitializer>*

<CondVarInitializer>::=
true | **false**

<CondArrayInitializer>::=
<*CondVarInitializer> [***{,<***CondVarInitializer>***}]**

2.6. Normale Variablen (*ordinary-variables*)

Normale Variablen beinhalten den Status des *coordinators*, der nicht direkt für die Sperrung kritischer Bereiche zuständig ist. Sie müssen auch nicht vom Typ *BOOL* sein. Häufig werden sie als Zählvariable oder ähnliches verwendet.

<VarDeclaration>::=
<PrimitiveType> <VariableDeclarator> [{, <VariableDeclarator>}] ;
<VariableDeclarator>::=
<Identifier> **=** *<VarInitializer>*
*<Identifier***[]>** **=** *<ArrayInitializer>*
<VarInitializer>::=
<Expression>
<ArrayInitializer>::=
<VarInitializer> [{,<VarInitializer>}]

2.7. Selbst-exklusive Methoden (*selfex*)

Self-exclusive-methods sind Methoden, die nicht mehrmals gleichzeitig ausgeführt werden dürfen. Wenn die Methode einmal gestartet wurde und von einem zweiten Thread ebenfalls ausgeführt werden soll, dann wird dieser Aufruf solang verzögert, bis die erste Ausführung beendet ist. Typische Beispiele sind Methoden, die eine private Membervariable einer Klasse ändern. Würden zwei Änderungen gleichzeitig ablaufen, so könnte das Ergebnis nicht den gewünschten Erfolg haben.

<SelfExclusiveMethods>::=
selfex **<***QualifiedName> [{,* **<***QualifiedName>}]* **;**

2.8. Mutual-exklusive Methods (*mutex*)

Mutual-exclusive-methods sind den *selfex* Methoden sehr ähnlich. Hier wird allerdings nicht eine einzige Methode für den exklusiven Gebrauch gesperrt, sondern eine ganze Anzahl von Methoden. Darum müssen bei der Deklaration von *mutex sets* auch mindestens zwei Methoden angegeben werden. Wird eine Methode aus diesem Set ausgeführt, so wird die Ausführung aller Methoden des Sets verschoben, bis die erste Methode beendet ist. Der typische Einsatz hierfür ist, wenn es mehrere Methoden gibt, die auf die gleiche(n) Variable(n) zugreifen und eine Parallelausführung wieder zu unspezifizierten Verhalten führen würde.
Methoden, die in einem *mutex set* deklariert wurden sind aber nicht selbst-exklusiv – d.h. es wird nur der Aufruf aller anderen Methoden des *sets* verhindert, nicht aber der erneute Aufruf der ersten Methode.

<MutuallyExclusiveMethodSet>::=
mutex *<QualifiedName> [{, <QualifiedName>}]*;

2.9. Methoden Manager

Method managers sind für die geschützten Bereiche (Verzögerungen) und die Benachrichtigung der Threads zuständig.

<MethodManager>::=
<QualifiedName> [{, <QualifiedName>}] :
[Requires]
[OnEntry]
[OnExit];

Die in *QualifedName* spezifizierten Methoden müssen Teil der koordinierten Klasse sein. Die nachfolgenden Funktionen erlauben es, die Methodenausführung zu schützen:

2.9.1. Geschützte Abschnitte (*guarded suspensions*)

Guarded suspensions werden bei COOL mit boolschen Variablen realisiert. Mit dem Schlüsselwort *requires* können dabei Zustände der Klasse abgefragt werden und nur wenn sie TRUE ergeben, wird die entsprechende Methode ausgeführt. Wenn die Abfrage FALSE ergibt, so wird der Aufruf so lange verschoben, bis er irgendwann TRUE wird.

2.9.2. *on_entry / on_exit*

Wie der Name schon sagt, handelt es sich hierbei um Funktionen, die beim Starten oder beim Beenden von Methoden ausgeführt werden.

2.10. Beispiel

```
public class BoundedBuffer {
  protected Object[] array;               // the elements
  protected int putPtr = 0, takePtr = 0;  // circular indices
  protected int capacity;
  protected int usedSlots = 0; // counter

  public BoundedBuffer (int size) {
    array = new Object[size]; capacity = size;
  }
  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    ++usedSlots;
  }
  public Object take() {
    Object old;
    old = array[takePtr];
    takePtr = (takePtr + 1) % array.length;
    --usedSlots;
    return old;
  }
}
```

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take};
  condition empty = true, full = false;

  put: requires !full;
       on_exit {
         if (empty) empty = false;
         if (usedSlots == capacity) full = true;
       }
  take: requires !empty;
       on_exit {
         if (full) full = false;
         if (usedSlots == 0) empty = true;
       }
}
```

Quelle: [CL1]

3. RIDL - Remote Interactions

3.1. Allgemeines

RIDL (*Remote Invokation Definition Language*) bietet Methoden an, die Daten über Rechnergrenzen hinweg zu übertragen. Bisher wurde immer angenommen, dass die Threads auf ein und derselben Maschine laufen. Ein echtes verteiltes System soll aber auch Rechenlast auf mehrere Rechner verteilen können. Hierfür ist es notwendig, dass Methoden von Klassen in Threads aufgerufen werden, die auf anderen Maschinen laufen. Man nennt dies Prozedurfernaufruf (*Remote Procedure Call – RPC*). RIDL bietet ein solches Konzept an. Der große Vorteil für den Entwickler ist, dass RIDL das völlig transparent anbietet. Der Entwickler kann seine Klassen so schreiben, als würden sie nur auf einer Maschine laufen. Er definiert sich zusätzlich noch ein „Portal“ über das dann die Fernaufrufe getätigt werden. Ein solches Portal ist nichts anderes als die Beschreibung, welche Methode von der anderen Maschine aus erreichbar sein soll und mit was für Datentypen sie arbeitet. Alles andere erledigt RIDL.

3.2. Portal Beschreibung (*portal body*)

Im *Portal body* werden die Methoden deklariert, welche von der Remote-Seite aus aufrufbar sein sollen. Außerdem kann noch die standardmäßige Transportart eingestellt werden (dazu später mehr).

<PortalBody>::=
{
<RemoteOperation> [{, <RemoteOperation>}]
[<DefaultTransfers>];
}
<DefaultTransfers>::=
default: *<TransferableType> [{<TransferableType>}]*

3.3. Remote Operations

Remote operations sind die Methoden, die von der Remote Seite aus aufgerufen werden dürfen. Der Methodenname muss ein gültiger Bezeichner innerhalb der verwalteten Klasse sein. Außerdem muss der Datentyp der Parameter und der des Rückgabewertes spezifiziert werden.

<RemoteOperation>::=
<ReturnType> <MethodName> **(** *[<Parameter> [{, <Parameter>}]]* **)**
[<RemoteOperationBody>];
<ReturnType>::=
<Type> |
void
<Parameter>::=
<Type> <ParamName>
<Type>::= (same as Java's *Type* production)

<ParamName>::=
<Identifier> |
<ParamName [ ]>

<RemoteOperationBody>::=
<ObjectTransferSpec> [{, <ObjectTransferSpec>}]

Die Übergabeparameter und die Rückgabewerte von Funktionen werden dabei nach folgenden Regeln übertragen:

1. primitive Datentypen werden immer kopiert
2. ist das Argument eine Referenz, so
    - gibt es eine Typ- oder Objektübertragungsspezifikation (3.4) - so wird das Objekt gemäß dieser Regel übertragen
    - die Standardregel ist, dass eine komplette rekursive Kopie des Objektbaumes übertragen wird, der das Objekt o als Wurzel hat

3.4. Type- und Objekt Transfer Spezifikationen

Hiermit wird spezifiziert, auf welche Art Argumente und Rückgabewerte übertragen werden sollen.

```
<ObjectTransferSpec>::=
        <ObjectName> : <Mode> ;
<ObjectName>::=
        <Identifier> |
        return
<Mode>::=
        gref |
        copy [<CopyDirective>]

<TypeTransferSpec>::=
        <ReferenceType> : <Mode> ;
```

3.5. Übertragung globaler Referenzen (*gref*)

Wird als Modus *gref* bei der Transferspezifikation gewählt, so wird das entsprechende Argument bzw. der Rückgabewert als globale Referenz übertragen. Es wird also nicht das Objekt sondern nur eine global gültige Referenz darauf übertragen. Wird das Objekt auf der Remoteseite angesprochen so wird dieser Aufruf wieder zurück an das eigentliche Objekt über das Portal hinweg geleitet, wo das Objekt ja existiert.

3.6. Übertragung von Kopien

Wird als Modus *copy* bei der Transferspezifikation gewählt, so wird das Argument bzw. der Rückgabewert als Kopie an die Remoteseite übertragen (bei primitiven Datentypen nur die Daten selber, bei Objekten alle verknüpften Objekte ebenfalls). Das kann aber auch zu Problemen führen. Ändert sich nach der Übertragung auf der lokalen Seite das Objekt, so werden diese Änderungen nicht an die Remoteseite übertragen. Andersherum ist es natürlich genauso. Die Kopie ist also immer nur ein Snapshot eines ganz bestimmten Zustandes zu einem Zeitpunkt.

3.7. Beispiel

Boundes Buffer:
Realisierung eines Datenpuffers fester Länge mit DJ und Java:

DJ

```
public class BoundedBuffer {
  private Object array[];
  private int putPtr = 0, takePtr = 0;
  private int usedSlots=0;

  public BoundedBuffer(int capacity) {
    array = new Object[capacity];
  }

  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    usedSlots++;
  }

  public Object take() {
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr + 1) % array.length;
    usedSlots--;
    return old;
  }
}
```

```
coordinator BoundedBuffer {
 selfex put, take;
 mutex {put, take};
 cond full = false, empty = true;
 put: requires !full;
   on_exit {
     empty = false;
     if (usedSlots == array.length)
          full = true;
   }
 take:  requires !empty;
     on_exit {
       full = false;
       if (usedSlots == 0) empty = true;
     }
}
```

JAVA

```
public class BoundedBuffer {
  private Object[] array;
  private int putPtr = 0, takePtr = 0;
  private int usedSlots = 0;

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
  }

  public synchronized void put(Object o) {
    while (usedSlots == array.length) {
      try {
        wait();
      }
      catch (InterruptedException e) {};
    }
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    }
    if (usedSlots++ == 0)
      notifyAll();
  }

  public synchronized Object take() {
    while (usedSlots == 0) {
      try {
        wait();
      }
      catch (InterruptedException e) {};
    }
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr+1) % array.length;
    }
    if (usedSlots-- == array.length)
      notifyAll();
    return old;
  }
}
```

Quelle: [CL1]

```
portal BoundedBuffer {
  void put(Book o);
  Book take();
 default:
    Book : copy { Book only all.String, all.int; };
}
```

Quelle: [CL1]

Dinning Philosophers:

Fünf „Philosophen“ sitzen an einem runden Tisch, essen oder philosophieren abwechselnd. Um zu essen, benötigen sie zwei Gabeln, die jeweils links und rechts neben ihnen liegen, die sie aber auch mit ihren Nachbarn teilen. Sie benötigen mehrere ‚Betriebsmittel’ um zu essen. Mögliche Probleme: Verklemmung oder Aushungerung.

DJ

```
class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static Fork forks[]= new Fork[max];
  static int count = 0;
  // for each philosopher
  protected int    mynumber;
  protected Fork   left, right;
  protected Random time = new Random ();

  Philosopher() { /* initialization */ }
  public void run() {/*loop: think, eat */}
  private void think() { /* think */ }

  private void eat() {
    left.take();
    right.take();
    int x = time.nextInt();
    try{
      Thread.sleep(Math.abs(x % 500));
    } catch(InterruptedException e){};
    left.put();
    right.put();
  }
}

per_class coordinator Philosopher {
  condition eating[max]=new boolean[false];
  eat: requires !eating[(mynumber+1)%max]&&
            !eating[(mynumber+max-1)%max];
       on_entry {
         eating[mynumber] = true;
       }
       on_exit {
         eating[mynumber] = false;
       }
}
```

JAVA

```
class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static Fork forks[]= new Fork[max];
  static int count = 0;
  static Object PhiLock = new Object();
  static boolean Eating[] = {false, false,
                             false, false, false};
  // for each philosopher
  protected int    mynumber;
  protected Fork   left, right;
  protected Random time = new Random ();

  Philosopher() { /* initialization */ }
  public void run() {/*loop: think, eat */}
  private void think() { /* think */ }

  private void eat() {
    synchronized (PhiLock) {
      while (Eating[(mynumber+1)%max] ||
             Eating[(mynumber+max-1)%max]){
        try {PhiLock.wait();}
        catch (InterruptedException e) {}
      }
      Eating[mynumber] = true;
    }
    left.take();
    right.take();
    int x = time.nextInt();
    try{
      Thread.sleep(Math.abs(x % 500));
    } catch(InterruptedException e){};
    left.put();
    right.put();
    Eating[mynumber] = false;
    synchronized (PhiLock) {
      Phi.notifyAll();
    }
  }
}
```

Quelle: [CL1]

4. Fallstudie

Es wurde ein D-Framework mit Java als Komponentensprache implementiert (DJ) [CL1]. Ferner wurden zehn typische algorithmische Probleme mit DJ und zum Vergleich in reinem Java implementiert. Die Lösungen in den beiden Sprachen wurden miteinander hin Hinblick auf Codegröße und Laufzeitverhalten verglichen.
In vielen Fällen wurde der Code mit Hilfe des D Frameworks kleiner und übersichtlicher.
Das Laufzeitverhalten ist zumeist schlechter als bei reinem Java. Hier könnten vielleicht Optimierungen noch helfen.

APPLICATION		DJ			ALTERNATIVE	%
#	NAME	JCORE	COOL+RIDL	TOTAL	IMPLEMENTATION	SMALLER
1	Bounded Buffer	48	16	64	64	0%
2	Philosophers	43	11	54	58	7%
3	Shape	24	5	29	48	40%
4	Matrix Multiplication	87	8	95	147	35%
5	Graph Traversal	92	12	104	104	0%
6	Assembly Line	108	25	133	152	13%
7	BookLocator/Printer	149	15	164	205	20%
8	Text Editor	215	15	230	232	1%
9	Document Service	246	12	258	369	30%
10	Message Queue	323	25	348	410	15%

Quelle: [CL2]

Laufzeitverhalten:

RIDL

1000 method invocations	
DJ	Java
no parameters: **10s**	*no parameters:* **10s**
one gref parameter: **24s**	*one parameter of type Remote:* **24s**
one copy parameter (object with 4 Integer fields): **26s**	*one parameter Serializable (object with 4 Integer fields):* **30s**
copying directive that selects 3 out of 4 Integer fields of a parameter: **12s**	*parameter with 3 Integer fields that is partially copied from an object of another class:* **8s**

Quelle: [CL2]

COOL

1000 method invocations per thread	
DJ	Java
Single thread calling a selfex method: **28ms**	*Single thread calling a synchronized method*: **7ms**
Two threads calling the same selfex method: **90ms**	*Two threads calling the same synchronized method:* **30ms**
Two threads calling 2 methods with requires, on_exit: **13s**	*Two threads, calling 2 methods with wait, notification:* **35s**

Quelle: [CL2]

5. Zusammenfassung & Ausblick

Verteilte Systeme stellen an die Fertigkeiten des Entwicklers auf Grund ihrer hohen Komplexität große Ansprüche. Das D Framework hat als Ziel, diese Probleme einfacher handhabbar zu machen. Es trennt die Lösung des algorithmischen Problems von der Problematik des verteilten Programmierens. Das macht den Code leichter verständlich und auch leichter zu schreiben. Der Entwickler kann sich auf sein eigentliches Ziel konzentrieren, ohne sich um das WIE des Verteilens kümmern zu müssen. Das geschieht zu einem anderen Zeitpunkt.
Leider scheint es außer der Dissertation [CL1] von Frau Christina Lopes kaum Bestrebungen zu geben, das D Framework zu benutzen.

6. Quellen

[CL1] Christina Lopes, “D: A Language Framework for Distributed Programming”
http://www2.parc.com/csl/groups/sda/publications/papers/Lopes-Thesis/

[CL2] Christina Lopes, [PowerPoint Präsentation]
http://www.ccs.neu.edu/research/demeter/course/f99/lectures/lec6-RIDL.ppt