Interaktive Ausführung von verteilten Algorithmen

Distributed Algorithms in Java (DAJ)

Mordechai Ben-Ari
Weizmann Institute of Science

Seminar zur Visualisierung verteilter Systeme

Christoph Jacob

Überblick

- Verteilte Algorithmen im Unterricht
- Anforderungen an Lernsoftware im Bereich verteilter Systeme
- Was ist DAJ ?
- Grundprinzipien von DAJ
- Demonstration von DAJ an Beispielen
- Beschreibung des Frameworks
- Hinzufügen eines Algorithmus
- Kritik und Lob
- Fazit

Verteilte Algorithmen im Unterricht

Nebenläufige verteilte Programme zu demonstrieren ist kompliziert, denn:

- Multiprozessor-Computer sind noch nicht sehr weit verbreitet
- Computer-Netzwerke sind zwar sehr verbreitet, aber Demonstration ist schwierig, denn:
  - Schwierigkeiten den exakten zeitlichen Ablauf zu planen
  - Schwierigkeit die Zustände aller Knoten simultan darzustellen
  - Netzwerkprogrammierung kompliziert und nicht portabel, daher für Hausarbeiten viel zu komplex

Anforderungen an Lernsoftware

- Algorithmus Schritt für Schritt ausführen und Szenarien erzeugen
- Die von den Knoten verwendeten Datenstrukturen simultan darstellen
- Minimale Hard- und Softwareanforderungen
- Möglichkeit einen neuen Algorithmus hinzuzufügen, ohne spezielle Programmierkenntnisse zu besitzen

Was ist DAJ ?

- Framework zum Schreiben von portierbaren verteilten Algorithmen in Java, die dann auf einem Rechner in einem Prozess visualisiert und interaktiv ausgeführt werden können
- Automatische Anzeige der internen Zustände der Knoten
- Komplette Kapselung von GUI- und Kommunikations-programmierung
- Systemvorraussetzungen: JRE 1.4 und eine Auflösung von mind. 800 x 600 (empfohlen 1024 x 768)

Konzeption von DAJ

- Alle Informationen über den Zustand des verteilten Systems werden in einem Fenster angezeigt.
- Der Schüler ist für die Zustandsänderungen des Systems verantwortlich, indem er den nächsten Schritt auswählt.
- Die möglichen Zustandsänderungen werden dem Schüler angezeigt, bei einigen Algorithmen auch teilweise nicht mögliche
- Die Möglichkeit der Protokollierung erlaubt es dem Schüler den Algorithmus neu zu starten und einen vorherigen Zustand wiederherzustellen (nicht bei Ausführung als Applet)

- Portables Java Programm, das entweder als Applikation oder als Applet ausgeführt werden kann
- Implementierung eines neuen Algorithmus benötigt nur allgemeine Java Programmierkenntnisse ohne Wissen über GUI Programmierung
- Einheitliche Anzeige unabhängig vom implementierten Algorithmus

Bereits vorhandene Algorithmen

- Byzantinische Generäle
- Gegenseitiger Ausschluss (Ricard-Agrawala)
- Verteilte Terminierung (Dijekstra-Scholten)
- Schnappschuss-Verfahren (Chandy-Lamport)
- Verteilte Terminationserkennung (Huang)
- Gegenseitiger Ausschluss (Suzuki-Kasami)
- Gegenseitiger Ausschluss (Lamport)
- Gegenseitiger Ausschluss (Maekawa subset)
- Byzantinische Generäle mit Absturz
- Byzantinische Generäle mit König

Byzantinische Generäle

Legende:

Kommunikationsproblem zwischen den osmanischen Generälen, die im Jahr 1453 n.Chr. Konstantinopel (das heutige Istanbul) belagerten. Wegen der starken Befestigung Konstantinopels war es notwendig, dass die Generäle mit ihren Truppen die Stadt gleichzeitig aus verschiedenen Richtungen angriffen.

- Generäle kommunizieren mit Boten miteinander, um sich auf eine gemeinsame Aktion zu einigen
- Einige der Generäle intrigieren gegen die anderen durch geschickte Fehlinformation

Satz für byzantinische Fehler (Lamport 1982):

Verteilte Übereinkunft angesichts byzantinische Fehler ist genau dann möglich, wenn von insgesamt 3m +1 Prozessoren höchstens m fehlerhaft sind.

Auf unsere Legende bezogen heißt das, dass die loyalen Generäle nur dann eine Einigungschance haben, wenn die Zahl der Intriganten kleiner als ein Drittel ist.

1. Phase: General entscheidet sich für einen Plan und teilt allen anderen Generälen seine Entscheidung mit

2. Phase: Die empfangenen Pläne werden an alle anderen Generäle weitergeleitet

1. Mehrheitsentscheidung, um zu entscheiden, was die Pläne der anderen Generäle gewesen ist

2. Mehrheitsentscheidung legt die weitere Vorgehensweise fest, also ob angegriffen wird oder nicht

Mehrheitsentscheidung benötigt 2/3 Mehrheit

Gegenseitiger Auschluss (Ricart-Agrawala)

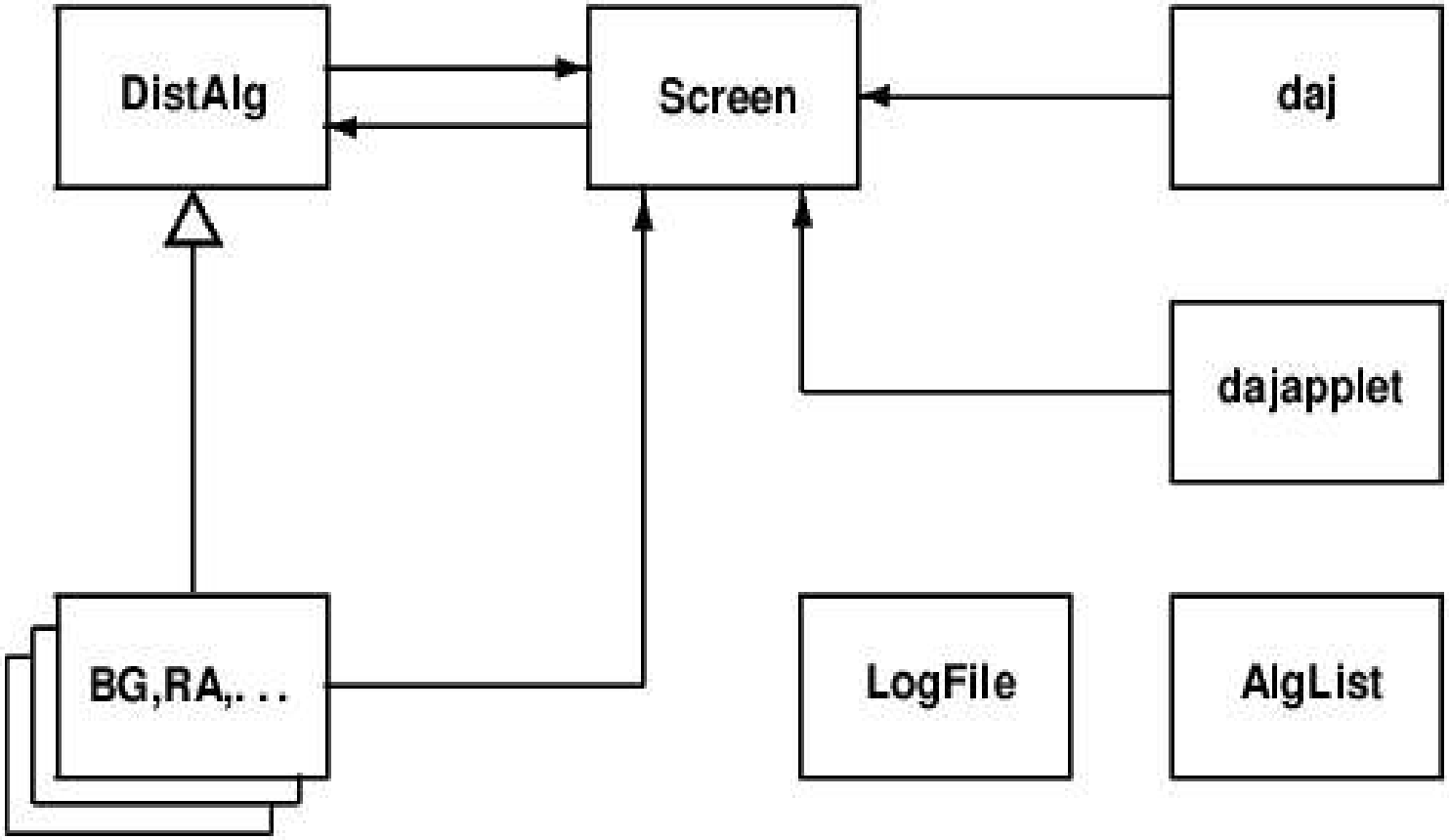
- Wenn ein Prozess in den kritischen Abschnitt eintreten will, zieht er eine neues Ticket mit einer höheren Nummer als er bereits empfangen hat und sendet an alle anderen eine Anforderung

- Wenn ein Anforderung empfangen wurde, wird eine Bestätigung sofort zurückgeschickt, wenn die eigene Ticketnummer höher ist als die empfangene oder der Prozess nicht in den kritischen Abschnitt eintreten möchte.

  Ansonsten wird die Bestätigung solange verzögert, bis der Prozess aus dem kritschen Abschnitt ausgetreten ist.

- Was passiert wenn beide Ticketnummern gleich sind ?

Framework



Algorithmus hinzufügen

Vorgehensweise:

- Klasse für den Algorithmus entwerfen, die von der abstrakten Klasse DistAlg erbt und die notwendigen Methoden implementiert (siehe nächste Folie)
- Eigentlicher Algorithmus wird mittels eines Zustandsautomaten realisiert (Methode *changeState*)
- Klassenname und Kürzel für den neuen Algorithmus in AlgList in den vorhandenen Code hinzufügen

Zu implementierende Methoden:

```
// Algorithm-specific initialization.
abstract protected void    init();

// Construct the button panels.
//   Calls addComponents for each button panel
//   and returns the initial button panel.
abstract protected int     constructButtons();

// Construct the text for the two title lines
//   and the two lines for each row.
abstract protected String  constructTitle1();
abstract protected String  constructTitle2();
abstract protected String  constructRow1(int row);
abstract protected String  constructRow2(int row);

// Implement the state machine.
abstract protected boolean stateMachine(String command);

// Receive a message sent by another node.
abstract protected void    receive(int message, int parm1, int parm2);
```

Methoden der Oberklasse, die verwendet werden können:

```
// Initialize the parent class after alg-specific initializaton.
protected void initialize()

// Create button panel, address button panel, return panel code.
protected int addComponents(String c1, String c2, String c3, String c4)
protected int addressButtons(String l, String special)

// Called from state machine to update state, button panel.
// The call changeState(newState, -2, newButtons)
// removes the old buttons and replaces with the new ones.
// (With this improvement, the second parameter is
// actually superfluous, but remains for historical reasons.
protected void changeState(int newState, int oldButtons, int newButtons)

// Send message to another node.
protected void send(int message, int to, int parm1, int parm2)

// Get node index from node name.
protected int FindNode(String b)
```

Alternating Bit Protokoll

Sender:

```
boolean bit = false;
do {
        if(ack.bit == bit) {
                data = produce();
                bit = !bit;
        }
        msg.send(data,bit);
} while(true);
```

Empfänger:

```
boolean bit = false;
do {
        if(msg.bit != bit) {
                consume(msg.data);
                bit = !bit;
        }
        ack.send(null,bit);
} while(true);
```

Quelltext (AlgList)

```
// Create and return object for an algorithm.
DistAlg getAlgObject(int index, int i, DistAlg[] da, Screen s)
{
  switch (index) {
      case  1: return new BG(i, da, s);
      ...
      case 11: return new AB(i, da, s);
    }
  return null; // Dummy statement for compiler
}

...

private static String[] algs =
  {"", "bg", "ra", "ds", "sn", "hg", "sk", "la", "ma", "cr", "kg","ab"};

private static final String[] titles = {
  "",
  "Byzantine generals",
  ...
  "Alternatin Bit Protocol"
};
```

Quelltext (AB)

```
package daj.algorithms;

import daj.DistAlg;
import daj.Screen;

// Alernating Bit Protocol Algorithmus
public class AB extends DistAlg {

    // Konstanten für Rollen der Knoten
    private static final int SENDER = 0;
    private static final int RECEIVER = 1;

    // Zustände
    private static final int READY = 0;
    private static final int WAITING = 1;
    private static final int AGAIN = 2;
    private static final int ACK = 3;

    // verwendete Datenstruktur (Bit)
    private boolean bit;

    // Buttons
    private int initButtons;
    private int againButtons;

    // Konstruktor
    public AB(int i, DistAlg[] da, Screen s) {
        super(i, da, s);
        init();
        initialize();
    }

    // Initialisieren des Bits und setzen
    // des Anfangszustandes
    protected void init() {
        bit = (me == SENDER) ? true : false;
        state = READY;
    }
```

```
// Buttons für die Interaktion des Benutzers erzeugen
protected int constructButtons() {
    if (me == SENDER) {
        initButtons =
            addComponents(
                "Nachricht Versenden: ",
                    "gestört",
                    "korrekt",
                    null);
        againButtons =
            addComponents(
                "N. erneut senden :",
                "gestört",
                "korrekt",
                null);
    }
    else {                          // me == RECEIVER
        againButtons =
            addComponents(
                "N. erneut Bestätigen: ",
                "gestört",
                "korrekt",
                null);
        initButtons =
            addComponents(
                "Nachricht Bestätigen: ",
                "gestört",
                "korrekt",
                null);
    }
    return 0;
}
```

```
// Name und Rolle des Knoten anzeigen
protected String constructTitle1() {
    if (me == SENDER)
        return Screen.node[me] + "(Sender)";
    else
        return Screen.node[me] + " (Empfänger)";
}

// Status des Knoten anzeigen
protected String constructTitle2() {
    if (me == SENDER) {
        switch (state) {
            case READY : return "Bereit neue Nachricht zu senden...";
            case WAITING : return "Warte auf Bestätigung";
            case AGAIN : return "vorherige Nachricht verlorengegangen, erneut senden...";
        }
    }
    else {    // me == RECEIVER
        switch (state) {
            case READY : return "Bereit neue Nachricht zu empfangen...";
            case AGAIN : return "vorherige Bestätigung verloren gegangen... erneut senden...";
            case ACK : return "Bestätigung senden...";
        }
    }
    return ""; // dummy für Compiler
}

// Zustand des Bits anzeigen
protected String constructRow1(int row) {
    return bit ? "Zustand des Bits: " + "1" : "Zustand des Bits: " + "0";
}
```

```
// Zustandsautomat des Algorithmus
protected boolean stateMachine(String command) {

    if (me == SENDER) {

        if (command.equals("korrekt")) {
            send(bit ? 1 : 0, (me + 1) % 2, 0, 0);
        }

        changeState(WAITING, -2, againButtons);
    }
    else {    // me == RECEIVER

        if (command.equals("korrekt")) {
            send(bit ? 1 : 0, (me + 1) % 2, 0, 0);
        }

        changeState(READY, -2, againButtons);
    }
    return true;
}
```

```
// empfangen einer Nachricht
protected void receive(int message, int parm1, int parm2) {

if (me == SENDER) {

        boolean ack = (message == 1) ? true : false;

        if (ack == bit) {
            bit = !bit;
            changeState(READY, -2, initButtons);
        }
        else changeState(AGAIN, -2, againButtons);
    }
    else {            // me == RECEIVER

        boolean msgBit = (message == 1) ? true : false;

        if (msgBit != bit) {
            bit = !bit;
            changeState(ACK, -2, initButtons);
        }
        else changeState(AGAIN, -2, againButtons);
    }
}
```

Kritik an DAJ

Benutzerfreundlichkeit:

- Kürzel für die verschiedenen Algorithmen sind als Benutzer nicht ersichtlich, damit ist das direkte Starten eines Algorithmus erschwert.
- Auch mit einer empfohlenen Auflösung von 1024 x 768 sind bei dem Algorithmus der byzantinischen Generäle mit 6 Generälen nicht alle Informationen zu sehen.
- Bereits implementierte Algorithmen teilweise schlecht beschrieben
- Verwendete Datenstrukturen werden bei manchen Algorithmen nicht angezeigt (z.B. bei Lamport)

- Meldung über einen nicht sinnvollen / relevanten Schritt wird dem Benutzer nur in dem Fenster für die Verfolgung der Ereignisse angezeigt, eigenes Fenster wäre sinnvoll

- Benutzeroberfläche recht unübersichtlich

Framework:

- In der Auswahlliste kann man nicht für einen Algorithmus, bei dem nur eine feste Anzahl von Knoten (z.B. Sender / Empfänger) erlaubt ist, die Auswahl des Benutzers einschränken.

- Kein Javadoc und teilweise schlecht dokumentiert

- Wenig bis gar kein Einfluss auf das Aussehen der Visualisierung

Lob an DAJ

- Strikte Kapselung von GUI- und Kommunikationsprogrammierung

  Dadurch ist das Hinzufügen eines weiteren Algorithmus sehr einfach und ohne spezielle Kenntnisse möglich

- Pädagogisch sehr gutes Konzept

  Interaktion des Schülers vermittelt aktives Wissen statt passives wie z.B. beim Zuschauen einer Animation

- GNU-Lizenz ermöglicht Weiterentwicklung von Dritten und Einsatz in Schulen / Universitäten ohne anfallende Lizenzskosten

Fazit

- DAJ konzentriert sich auf die am meisten problematischen Aspekte beim Unterrichten von verteilten Systemen wie zum Beispiel die Zustände des Systems und die Daten-strukturen der Knoten zu verstehen
- angezeigte Informationen ermöglichen Diskussionen und Untersuchungen der Schüler
- Interaktivität erlaubt vor allem jungen Schülern an einer Oberschule einen einfachen Zugang zu der Materie, ohne einer formalen und abstrakten Behandlung der Algorithmen, wie sie an der Universität geschieht.

Fazit

- Interaktion als gutes pädagogisches Mittel im Unterricht, fördert aktives statt passives Wissen
- Leichte Implementierung von zusätzlichen Algorithmen
- Das Konzept der einheitlichen Oberfläche hat eine etwas starres Format zur Folge
- Keine Animation, daher manchmal nicht besonders anschaulich