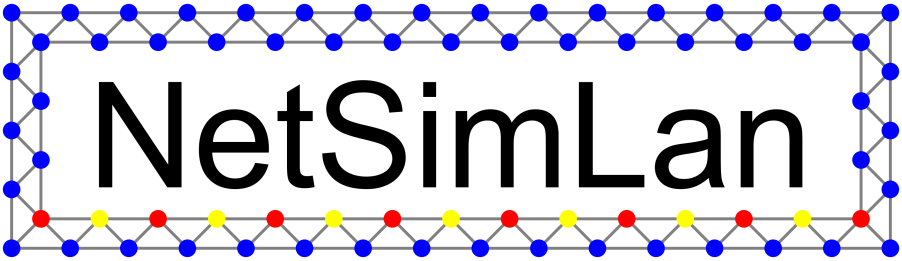
UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft



Handbuch

Björn Beckendorf

NetSimLan ist eine Software zur Simulation verteilter Systeme. Sie ist insbesondere für die Visualisierung von Selbststabilisierungsalgorithmen geeignet, aber nicht auf diese Problemart beschränkt. Mit NetSimLan ist es möglich, Algorithmen in einer einfachen Sprache, die wir NetSimLan-Sprache nennen, zu programmieren. Kommunikation zwischen Knoten wird mit dieser NetSimLan-Sprache abstrakt dargestellt, wie ich in Kapitel 2.3 vorstellen werde. Diese Software hat Maßnahmen zur Fehlerbehandlung von nicht gesetzten Variablen, fehlerhaften Arrayindizes und arithmetischen Fehlern. Näheres zur Fehlerbehandlung ist jeweils bei den betreffenden Beschreibungen zu finden. Das Programm wird übersetzt und anschließend als Simulation ausgeführt. NetSimLan bietet semisynchrone und asynchrone Kommunikation sowie explizite Synchronisierung. Mehr dazu folgt in Kapitel 4.

1 Installation und Aufruf

1.1 Installation

Es gibt verschiedene Möglichkeiten, NetSimLan zu nutzen.

Linux

Unter NetSimLan Ubuntu und Debian kann NetSimLan in das System integriert werden, sodass es zusammen mit dem Rest des Betriebssystems aktuell gehalten wird. Hierzu genügt es, diesen Befehl auszuführen:
`wget -O nslsetup https://apt.netsimlan.org/setup && bash nslsetup`

Auf anderen Linux-Distributionen muss zunächst das `Java Develoment Kit` in Version 8 oder höher und `bash` installiert sein. Meist ist `bash` bereits vorinstalliert. Java kann über die jeweilige Paketverwaltung installiert werden, hierzu sei auf die jeweiligen Dokumentationen der Distributionen verwiesen.
Mit diesen Voraussetzung muss der Nutzer nur die Datei `netsimlan-linux.tar.xz` entpacken. Um NetSimLan optional fest zu installieren, kann Programm einfach nach `/usr/local` kopiert werden.

Windows

Zunächst muss das `Java Develoment Kit` in Version 8 oder höher installiert werden. Siehe dazu die offiziellen Java-Webseiten. Danach muss lediglich die Datei `netsimlan-windows.zip` entpackt werden.

1.2 Aufruf

Gestartet wird NetSimLan aus dem Terminal. Hierzu muss `netsimlan myFile` aufgerufen werden. `myFile` ist hierbei die Datei mit dem eigenen Programm, das simuliert werden soll. Falls NetSimLan nicht in einem Standardverzeichnis installiert ist und daher nicht gefunden wird, muss der ganze Pfad angegeben werden.

Es können weitere Parameter angehängt werden: `-f Datei` lädt einen initialen Graph (siehe 5). Mit `-l Datei` kann die Simulationsausgabe in einer Datei gespeichert werden. Mit `-g off` und `-v off` lässt sich die grafische Oberfläche und die Visualisierung ausschalten. `-t s` begrenzt die Simulation auf s Sekunden und gibt eine ausführliche Statistik am Ende der Simulation aus. Ferner lässt sich mit `-i s` das Timeout-Intervall

einstellen (genaueres zu timeouts siehe 2.3). Diese Parameter ermöglichen automatisierte Experimente. Für Windows muss die Datei `Launcher.jar` aufgerufen werden, der alle Parameter grafisch entgegen nimmt und mit dieselben Programmparametern wie der Befehl `netsimlan` akzeptiert.

Zunächst verarbeitet bei einem Aufruf der Übersetzer die Eingabe. Um dies zu ermöglichen, muss das eigene Programm in einem beschreibbaren Verzeichnis liegen. Andernfalls kann der Übersetzer das Zwischenergebnis nicht speichern. Falls bei dieser Verarbeitung Fehler in der Syntax oder statischen Semantik festgestellt werden, erscheinen diese im Terminal. Falls dies nicht der Fall ist, wird eine grafische Oberfläche gestartet, von der aus der Nutzer die Simulation steuert.

2 Programmiersprache von NetSimLan

Die Programmiersprache von NetSimLan, stellt unter anderem folgende Elemente zur Verfügung.

- Variablen der Typen `node`, `int`, `float`, `bool`, `string`, Arrays und Hashtabellen. (2.4)
- Informationsabfrage zu Variablen: `length`, `id`, `type` und 3 Varianten von `hash` (2.3)
- Ausdrücke einschließlich arithmetischer und boolescher Operationen (2.5)
- Schleifen `for` und `while` (2.6)
- Verzweigungen `if` (2.6)
- verschiedene Knotenarten in einem Programm (2.2)
- Methoden mit und ohne Parameter, aufrufbar lokal und extern (2.3)
- eingebaute Methoden `print`, `sleep` und `kill` (2.3)
- verschachtelte Blöcke `{}`

In Klammern ist jeweils das Kapitel angegeben, in dem das Thema näher erläutert wird. Blöcke können überall dort genutzt werden, wo ein Befehl steht. Als Befehle in diesem Sinn gelten Deklarationen, Zuweisungen, Blöcke, Funktionsaufrufe, Schleifen und Verzweigungen. Im Umfeld von Programmiersprachen werden Leerzeichen, Zeilenumbrüche und Tabulatoren auch als *Whitespaces* bezeichnet. Wichtig ist, dass Whitespaces außerhalb von Strings lediglich zur Trennung von Bezeichnern und Schlüsselworten dienen. Sie haben keinen Einfluss auf die Struktur des Programms. Daher kann der Programmierer seinen Quelltext so strukturieren, wie es für ihn am sinnvollsten ist. Ferner kann der Nutzer Kommentare zwischen Quelltextelementen platzieren. Kommentare haben die Form `/* mein Kommentar */`.

Die vereinfachte Grammatik der NetSimLan-Sprache ist wie in Abbildung 1 aufgebaut. Diese Schreibweise der Regeln sind folgendermaßen zu lesen: `A:B.` bedeutet, dass `A` zu `B` abgeleitet werden kann. Jede Ableitungsregel endet mit einem Punkt. Verschiedene Ableitungsregeln für ein Nichtterminal sind Alternativen zueinander. `A/B` steht für `A` oder `B`. Zusammen ist `A:B.A:C.` mit `A:B/C.` gleichbedeutend. Die Kombination `A*` bedeutet, dass `A` beliebig häufig wiederholt werden kann. `A+` bedeutet, dass `A` mindestens einmal vorkommt, ist also die Kurzform für `A A*`. Die Regel `A B` steht für `A` gefolgt von `B`. *Identifier*, *Number* und *String* sind Terminale, ich später genauer erläutere. Zeichen oder Zeichenketten zwischen ' bedeuten,

Program	:	NodeBlock + .
NodeBlock	:	Identifier '{' Declaration * Function* '}' .
Block	:	'{' Statement * '}' .
Statement	:	Block / Declaration / Ass ';' / FunCall / LoopIf /
		'return' ';' / 'return' Expression ';' / 'break' ';' / 'continue' ';' .
Declaration	:	(Type ['=' Expression] / Type ('[]' / '<>')+ / '[' Type ',' Type ']')
		Identifier ['ignore'] [VarColorView] .
VarColorView	:	'blue' / 'red' / 'green' / 'cyan' / 'magenta' / 'yellow' .
Type	:	'node' / 'int' / 'float' / 'string' / 'bool' .
Variable	:	Identifier / Identifier ('[' Expression ']')+ / 'this' .
Ass	:	Variable AssOpr Expression / Variable '++' / Variable '–' /
		Identifier '>>' Expression /
		Identifier '>[' Expression ']>' Expression /
		Identifier '<<' / Identifier '<[' Expression ']<' .
AssOpr	:	'=' / '+=' / '-=' / '*=' / '/=' / '%=' / '&=' / '| =' .
FunCall	:	[Variable '->'] FunctionNameUse '(' ExpressionList ')' ';' .
ExpressionList	:	ExpressionList ',' Expression / Expression / .
FunctionNameUse	:	'print' / 'visLabel' / 'sleep' / 'kill' / 'mark' / 'sync' / Identifier .
Expression	:	Expression BinOpr Expression / UnOpr Expression / Operand .
BinOpr	:	'||'/'&&'/'|'/'^'/'&'/'=='/'!='/
		'<'/'<='/'>='/'>'/'.'/'<<'/'>>'/'>>>'/'+'/'-'/'*'/'/'/'%'/'^^'.
NegOpr	:	'-' / '!' .
Operand	:	'(' Expression ')' / Number / 'true' / 'false' / Variable /
		AttributName '(' Variable ')' / String / 'null' .
AttributName	:	'length' / 'id' / 'longitude' / 'latitude' / 'type' / 'hash' / 'hash1' / 'hash2'
		/ 'containts' / 'str_len' / 'sub_str' / 'random' / 'sqrt' / 'log' / 'round' .
LoopIf	:	'for' '(' [Ass] ';' Expression ';' [Ass] ')' Statement /
		'for' '(' Type Identifier : Expression ')' Statement /
		'for' '(' Type Identifier : Type Identifier : Expression ')' Statement /
		'while' '(' Expression ')' Statement /
		'if' '(' Expression ')' Statement ['else' Statement] .
Function		Identifier '(' ParamList ')' Block .
ParamList	:	ParamList ',' ParamDeclaration / ParamDeclaration / .
ParamDeclaration	:	Type Identifier / Type '[]' Identifier .

Abbildung 1: Vereinfachte Grammatik der NetSimLan-Sprache.

dass dieses Zeichen beziehungsweise diese Zeichenkette genau in dieser Form an dieser Stelle auftaucht. Sie gehören folglich zu den Terminalen.

Bezeichner (engl. *Identifier*) jeder Art folgen dem regulären Ausdruck `[a-zA-Z_][a-zA-Z_0-9]*`. Jedoch darf ein Bezeichner keines der in der Grammatik genannten Schlüsselwörter sein. Ganze Zahlen (engl. *Number*) können dezimal, hexadezimal (mit führendem `0x`) oder oktal (mit führender `0`) angegeben werden. Fließkommazahlen werden mit Dezimalpunkt angegeben und können mit Zehnerpotenzen durch ein angehängtes `e`n für $\cdot 10^n$ angegeben.
Der komplette reguläre Ausdruck ist `(([0-9]+\.[0-9]*)|(\.[0-9]+))((e|E)(\+|\-)?[0-9]+)?)|([0-9]+(e|E)(\+|)?[0-9]+)` Zeichenketten (engl. *String*) werden durch Anführungsstriche begrenzt, beispielsweise `"Text"`.

2.1 Beispiele

Um einen Eindruck von der NetSimLan-Sprache zu bekommen, steht ein Paket mit verschiedenen Beispielprogrammen zum Download bereit. Diese werden in diesem Kapitel kurz vorgestellt.

Das Beispiel `showStructure` zeigt an, wie der Graph initial aufgebaut wurde. Die möglichen initialen Graphen werden in Kapitel 3.1 näher beschrieben. Das Beispiel speichert hierbei alle Kanten, die erzeugt werden, in ein Array.

Das zweite Beispiel `broadcast` besteht aus einen zentralen Server und beliebig vielen Clients. Der Anwender sollte zuerst den Server erzeugen und anschließend die Clients. Bei den Clients sollte der Nutzer einstellen, dass diese direkt mit dem Server verbunden werden (siehe Kapitel 3.1). Nachdem das Netzwerk für das Beispiel aufgebaut wurde, kann ein Client mit `broadcast("hello");` einen Text an den Server senden. Dieser sendet den Text an alle ihm bekannten Knoten weiter, welche es auf der Konsole ausgeben. Dieses Beispiel ist sehr einfach aufgebaut, enthält aber die wichtigsten Elemente der NetSimLan-Sprache. Deswegen eignet es sich, um einen Eindruck der NetSimLan-Sprache zu geben. Der Quelltext zu diesem Beispiel ist in Abbildung 2 zu finden. Es ist hierbei bewusst eine Formatierung gewählt, die an verschiedenen Stellen variiert. Dies verdeutlicht die Freiheiten in der Formatierung, etwa in der Einrückung oder in Zeilenumbrüchen.

```
1  Server { node [] clients;
2    entry(node C){
3      int i;
4      int len;
5      len=length(clients);
6      for(i=0;i<len;i++)
7        if(clients[i]==C){
8          return;
9        }
10     clients>>C;
11     C -> entry(this);
12   }
13   broadcast(string M){
14     int i;int len;
15     len=length(clients);
16     for (i=0;i<len;i++){
17       clients[i] -> write(M);
18     }
19   }
20 }
21 Client {
22   node server;
23   entry(node S){
24     if (server == null){
25       server = S;
26       server -> entry(this);
27     }}
28   write(string M){print(M);}
29   broadcast(string M){
30     server -> broadcast(M);
31   }
32 }
```

Abbildung 2: Der Quelltext zum Beispiel `broadcast`.

Bei `build-list` handelt es sich um einen selbststabilisierenden Algorithmus. Dieser Algorithmus baut aus einem beliebig schwach zusammenhängenden Graphen eine geordnete Liste. Es handelt sich um den Algorithmus aus der Vorlesung „Verteilte Algorithmen und Datenstrukturen“. In diesem Vorgehen hat jeder Knoten einen linken und einen rechten Nachbarn. Die Knoten reichen Verbindungen weiter, sofern diese nicht zu einem direkten Nachbarn gehören. Außerdem stellen sie sich periodisch gegenseitig vor, um sicherzustellen, dass die Liste in beide Richtungen geordnet ist und jeder Knoten an der richtigen Position in der Liste ist.

Als letztes Beispiel existiert `grid-message`. Hierbei handelt es sich um ein einfaches Gitter. Es ist möglich, Nachrichten von einem Knoten zu einem anderen Knoten zu senden. Hierbei wird der Knoten markiert.

Um die Markierungen zu entfernen muss auf Knoten Nummer 1 die Methode `unmark()` aufgerufen werden. Wichtig ist, dass die Größe, die im Quelltext angegeben ist, zu der Größe des erstellten Gitters passt. Standardgröße ist 5, hierfür müssen $5^2 = 25$ Knoten erstellt werden. Für die Standardgröße liegt ein fertiger Graph unter dem Dateinamen `grid-message-graph` bei. Es handelt sich bei dem Beispiel um kein selbststabilsierendes System. Daher muss der Nutzer bei der Erstellung des Graphen darauf achten, die richtige Struktur zu wählen. Näheres dazu folgt in Kapitel 3.1.

2.2 Knotentypen

Im Weiteren wird nun der Aufbau von Programmen für NetSimLan genauer erläutert. Ein Programm besteht aus mindestens einem Knotentyp, der einen Namen trägt. Sollen verschiedene Knotentypen verwendet werden, können diese ohne Weiteres hintereinander geschrieben werden. Der Aufbau eines Knotentyps mit dem beispielhaften Bezeichner `name` ist dabei wie folgt.

```
1 name{
2   Attribute
3   Methoden
4 }
```

Hierbei ist `Attribute` eine möglicherweise leere Liste von Variablendeklarationen und `Methoden` eine möglicherweise leere Liste von Methodendefinitionen. Zu Variablen siehe 2.4, zu Methoden siehe 2.3. Der Typ eines Knotens muss eindeutig sein. Er kann abgefragt werden, wie ich in Kapitel 2.3 erläutere.

2.3 Methoden

Methoden sind wie folgt aufgebaut:

```
1 name type(type p1, type p2){ Befehle }
```

Dabei stellt `name` den Namen dieser Methode dar. Der Name ist ein Bezeichner und muss innerhalb eines Knotentyps eindeutig sein. Falls ein Methodenname in verschiedenen Knotentypen verwendet wird, sollte darauf geachtet werden, dass die Methoden in allen Knotentypen dieselbe Signatur haben. Andernfalls kann es zu Problemen beim Aufruf der Methode kommen, falls sie von einem anderen Knoten aus aufgerufen werden soll (s.u.).

Im schematischen Methodenaufbau sind `type p1, type p2` die Parameter der Methode. Es können beliebig viele durch Kommata getrennte Parameter angegeben werden. Dabei werden Typen, Hashtabellen und Arrays so definiert, wie es auch bei Variablen im Allgemeinen der Fall ist (vgl. 2.4). Arrays und Listen (Unterschied erläutere ich in Kapitel 2.4) können gegeneinander ausgetauscht werden.

Es gibt vier Methoden, die eine besondere Stellung einnehmen und selbst implementiert werden können: `init()` ist der Konstruktor eines Knotens und wird gemäß Modell beim Start des Knotens ausgeführt. `timeout()` wird nach dem Modell regelmäßig nach Ablauf einer bestimmten Zeit aufgerufen. Diese ist standardmäßig 100 ms, wobei dieser Wert während der Simulation verändert werden kann. Um zu simulieren, dass die Knoten nicht synchron arbeiten, wird dieser Wert bei jeder Wartezeit mit einem Zufallswert $\pm 10\%$ variiert, wobei der Grundwert nicht verändert wird. Die dritte Methode dieser Art ist `entry(node n)`. Diese wird von der Simulationsumgebung aufgerufen, um einen Graphen zu bilden. Falls sie nicht implementiert

ist, kann die Simulationsumgebung nur einzelne Knoten erstellen, sie jedoch nicht miteinander verknüpfen. Näheres dazu, wie der Anwender Knoten in der Simulation erstellt und wie sie dabei initial verknüpft werden, folgt in Kapitel 3.1. Die Methode `startRound()` wird zur Synchronisierung genutzt, was in Kapitel 4 näher beschrieben ist.

Der Aufruf einer Methode erfolgt entweder lokal durch `name(p1,p2);` oder entfernt durch `n->name(p1,p2);`, wobei `n` eine Variable vom Typ `node` ist. Näheres zu Variablen und Datentypen folgt in Kapitel 2.4. Jede selbst geschriebene Methode kann mit beiden Varianten aufgerufen werden. Ein lokaler Aufruf erfolgt immer im selben Knoten. Er wird direkt verarbeitet und kehrt nach dessen Beendigung zur aufrufenden Methode zurück. Ein entfernter Aufruf hingegen wird an den entfernten Knoten gesendet und dort später, bei dessen nächstem Timeout, abgearbeitet. Möchte man dieses Verhalten für einen lokalen Methodenaufruf erreichen, so kann man diesen in einen entfernten Aufruf umwandeln, indem dort `n` durch `this` ersetzt wird. Ein Knoten verarbeitet immer nur eine Operation gleichzeitig. Das bedeutet, dass er weder lokale noch entfernte Methodenaufrufe nebenläufig verarbeitet.
Die Parameter müssen im Kontext des Aufrufs gültige Ausdrücke oder Variablen sein. Sollte `n` dabei `null` sein, wird der Aufruf verworfen. Dieses Verhalten erspart eine vorherige Prüfung auf diesen Umstand und ermöglicht dem Anwender eine einfache Umsetzung der Kommunikation in seinem Algorithmus. Falls man hierbei eine Kante an einen anderen Knoten weiterreicht und bei dem Aufrufer entfernt, sollte man hierbei aufpassen, den Zusammenhang des Graphen nicht zu verlieren. Falls bei einem entfernten Aufruf `n->name(p1,p2);` die Methode `name` im Knoten `n` nicht implementiert ist, ignoriert `n` den Aufruf.

Mit `return Expr;` ist es möglich, Methoden zu verlassen. Methoden können einen Rückgabewert vom Typen `type` in der Signatur haben, die nur bei lokalen Aufrufen zurückgegeben werden. Diese werden als Ausdruck für `Expr` übergeben. Falls kein `return` angegeben wird, gibt die Methode den Standardwert des jeweiligen Typen zurück (mehr dazu unter 2.4) Wenn die Methode keinen Rückgabewert haben soll, wird `type` und entsprechend `Expr` weggelassen. Es existieren folgende vordefinierte Methoden: Die erste Methode ist `sleep()`, die den Knoten bis zum Eintreffen der nächsten Nachricht an ihn schlafen legt. `kill()` beendet den Knoten. Er benachrichtigt dabei seinen Nachbarn nicht und führt keine weiteren Aktionen aus bearbeitet keine Nachrichten. Die dritte vordefinierte Methode ist `print(o)`, die eine Ausgabe auf der Konsole vornimmt. Die Methode nimmt jeden beliebigen Datentyp, jedoch keine Arrays oder Hashtabellen, entgegen. Ähnlich zu `print` arbeitet `visLabel(o)`, das die Beschriftung des Knotens in der Visualisierung setzt. `visLabel(null)` setzt die Beschriftung auf den Namen zurück. `length(a)` ermittelt, wie viele Elemente in dem Array `a` sind. Mit `id(n)` lässt sich der eindeutige Identifikator eines Knotens `n` ermitteln. Er wird bei jeder Simulation von 1 beginnend aufsteigend an die Knoten verteilt. Mit `type(n)` lässt sich der Typ des Knotens ermitteln. Genaueres zu Knotentypen findet sich unter 2.2. `longitude(n)` und `latitude(n)` ermitteln die Position des Knotens als *float*. Mit `hash(o)`, `hash1(o)` und `hash2(o)` lassen sich Prüfsummen zu einem Objekt `o` ermitteln. Dabei können die Methoden jeden beliebigen Typen verarbeiten. Die drei Versionen unterscheiden sich in den Werten, die sie zurückgeben, jedoch nicht im sonstigen Verhalten. Dies kann für Fälle genutzt werden, in denen man Prüfsummen für verschiedene Zwecke benötigt. Eine Fließkommazahl f lässt sich mit `round(f)` auf die nächste Ganzzahl runden. Dabei wird bis $0,4\ldots$ abgerundet und ab $0,5$ aufgerundet. Für eine Hashtabelle `h` kann mit `containts(h,k)` ermitteln, ob ein Wert für den Schlüssel `k` enthält. Möchte man den Identifikator, den Typen oder die Prüfsumme des aktuellen Knotens ermitteln, so setzt man für `n` beziehungsweise `o` den Ausdruck `this` ein. `type(n)` gibt eine Zeichenkette, die anderen beschriebenen Methoden eine Zahl zurück. Sie können wie Variablen in Ausdrücken genutzt werden.

Für arithmetische Operationen gibt es auch die Wurzelfunktion `sqrt(i)`, die die Wurzel von `i` ermittelt, `log(i)` für den Logarithmus zur Basis 2 und `random(i)`, die eine ganzzahlige Zufallszahl im Intervall $[0, i)$ zurück gibt. Für die Länge eines Strings `s` steht `str_len(s)` zur Verfügung. Mit `sub_str(s,a,b)` lässt sich ein Teilstring zwischen `a` und `b` ermitteln. Beispielsweise ist `sub_str("Hello World!", 6, 11)` im Ergebnis `World`.

2.4 Datentypen, Variablen und Zuweisungen

Variablen und Attribute gelten in dem kleinsten sie umgebenden Block sowie in allen Blöcken darin. Sie gelten ab der Zeile ihrer Deklaration. Innerhalb eines Blocks kann ein Variablenname nicht mehrfach deklariert werden, jedoch kann derselbe Variablenname in verschachtelten Blöcken in jedem Block jeweils einmal neu deklariert werden. Sollte in verschachtelten Blöcken mehrmals derselbe Variablenname definiert sein, so gilt bei der Verwendung jener, der in dem kleinsten umgebenden Block definiert ist. Variablen können einzeln deklariert werden mit `type name;`, als Array mit `type [] name;`, als Liste mit `type <> name;` und als Hashtabelle mit `[type,type] name;`. Dabei ist `type` einer der folgenden Typen: `node` zum Speichern von Verbindungen, `int` für ganze Zahlen, `float` für Fließkommazahlen, `bool` für Wahrheitswerte und `string` zum Speichern von Zeichenketten. Mehrdimensionale Arrays werden deklariert, indem das [] bzw. <> mehrfach geschreiben wird, ein dreidimensionales Array sähe beispielsweise wie folgt aus: `type [][][] name;`. Jede Variable hat zu Beginn einen Standardwert. Bei einem `node` ist dieser `null`, bei einem `int` ist er `0`, bei einem `float` ist er `NaN`, bei einem `bool` ist er `false` und bei einem `string` ist er eine leere Zeichenkette. Arrays und Hashtabellen ist zu Beginn leer. Einer einzelnen Variable kann ein Wert bei der Deklaration mit `type name = expression;` festgelegt werden.

In der Visualisierung, die in Kapitel 3.3 vorgestellt wird, werden die Knoten gemäß ihrer Kanten zu anderen Knoten angeordnet. Wenn das für eine bestimmte Kante verhindert werden soll, wird diese mit dem Schlüsselwort `ignore` vor dem Semikolon ergänzt. Zur übersichtlichen Gestaltung der Visualisierung ist es dem Nutzer möglich, Kanten einzufärben. Es stehen folgende Farben zur Verfügung: `blue`, `red`, `green`, `cyan`, `magenta`, `yellow`. Diese werden anstelle beziehungsweise hinter `ignore` gesetzt. Um eine Kante komplett auszublenden, kann die "Farbe"`hidden` verwendet werden. Eine Kante mit beiden Optionen ist beispielsweise `node myEdge ignore green;`. Diese beiden Optionen stehen nur für Variablen des Typs `node` zur Verfügung. Ferner darf es sich bei der Variable nicht um eine solche handeln, die nur innerhalb einer Methode existiert.

Listen und Arrays unterscheiden sich in sofern, als dass eine Liste als verkettete Liste, ein Array als Arraylist angelegt wird. Im weiteren unterscheiden sie sich nicht in der Handhabung und werden daher unter dem Wort „Array“ zusammengefasst. Bei Funktionsparametern können diese beiden Typen vertauscht werden.

Ein Auslesen des Wertes erfolgt durch Angabe des Namens der Variable oder, im Falle eines Arrays oder einer Hashtabelle, durch Angabe des Namens gefolgt von der Position in eckigen Klammern, beispielsweise `name[1]` oder `name[1][5][10]`. Indizes für ein Array mit `n` Elementen beginnen bei 0 und gehen bis $n-1$, wobei n die Länge des Arrays ist. Sollte in einem Array die angegebene Position nicht existieren, wird der Wert zurückgegeben, der initial für diesen Datentyp vorgesehen ist (siehe oben).

Eine einfache Zuweisung ist `v=w;`, welche `w` in `v` speichert. Für Arrays und Hashtabellen ist eine einfache Zuweisung `a[i]=w;`, welche `w` in `a[i]` speichert und den alten Wert dabei überschreibt. Hierbei ist `v` eine

einfache Variable, `a` ein Array bzw. eine Hashtabelle und `i` ein Ausdruck, der als Positionsangabe in einem Array bzw. der Hashtabelle dient. Dieser Ausdruck muss zu dem Indextyp von `a` passen, insbesondere ist dieser bei einem Array eine Zahl. `w` ist der Ausdruck, dessen Wert gespeichert werden soll. Dabei muss `w` zum Typ von `v` bzw. `a` passen. Es ist möglich einen `int` in einen `float` zu speichern. Dieser Wert wird automatisch umgewandelt. Falls die Typen nicht zueinander passen, wird das zum Zeitpunkt der Übersetzung automatisch angemerkt. Sollte das Array bei der Zuweisung zu klein sein, um den Index `i` zu haben, werden entsprechend viele Elemente hinzugefügt, bis `a` genügend Elemente enthält, damit wird `w` in `a` an der Stelle `i` gespeichert werden kann.

Um mit Arrays als Liste umgehen zu können, bietet die NetSimLan-Sprache spezielle Varianten von Zuweisungen. `a>>w;` fügt `w` am Ende zu `a` hinzu und erweitert `a` entsprechend. `a<<;` löscht das letzte Element aus `a` und verkleinert `a` entsprechend. `a>[i]>w;` fügt `w` an Index `i` zu `a` hinzu, erweitert `a` und verschiebt alle bisherigen Elemente ab Index `i` entsprechend. `a<[i]<;` löscht das Element an der Stelle `i`, verschiebt alle folgenden Elemente und verkleinert das Array entsprechend. Mit `a<[i]<;` können ebenfalls aus einer Hashtabelle Werte gelöscht werden. Ein Array und eine Hashtabelle kann mit `a<<<<;` komplett geleert werden. All diese Operationen sind auf eindimensionale Arrays beschränkt.

Ein `int` wird mit `v++;` inkrementiert und mit `v--;` dekrementiert. In einem Array oder einer Hashtabelle nutzt man entsprechend `a[i]++;` oder `a[i]--;`. Im Fall eines Arrays wird der Befehl nicht ausgeführt, falls `i` außerhalb von 0 und $length(a)$ liegt. Darin unterscheidet sich `a[i]++;` von `a[i]=a[i]+1;` . Letzteres hängt nach oben beschriebenem Verhalten eine `1` an `a` an.

Ein eindimensionales Array kann man mit den beschriebenen Zuweisungen auch als Schlange (englisch *Queue*) oder Stapel (englisch *Stack*) verwenden. In beiden Fällen wird ein neues Element w mit `a>>w;` hinzugefügt. Bei einem Stapel wird das oberste Element mit `a[length(a)-1]` abgefragt und mit `a<<;` entfernt. Entsprechend wird bei einer Schlange das vorderste Element mit `a[0]` abgefragt und mit `a<[0]<;` entfernt.

Für das Konstrukt `v=v + w;` kann die Kurzform `v+=w;` verwendet werden. Entsprechende Kurzformen existieren für `+=`, `-=`, `*=`, `/=`, `%=`, `&=` und `|=`. Näheres zu Ausdrücken folgt in Kapitel 2.5.

2.5 Ausdrücke

In NetSimLan können Ausdrücke mit arithmetischen und booleschen Operationen beschrieben werden. Darüber hinaus ist es möglich Zeichenketten zu verknüpfen. Es gibt folgende Operationen mit zunehmender Bindung, wobei `A` und `B` selbst bereits Ausdrücke des jeweils passenden Typs sind:

1. abbrechendes oder `A || B`
2. abbrechendes und `A && B`
3. oder `A | B` und exklusives oder `A ^ B`
4. und `A & B`
5. Gleichheit und allgemeine Ungleichheit `A == B`, `A != B`
6. Vergleich zwischen Zahlen `A < B`, `A <= B`, `A > B`, `A >= B`

7. Shiften von ganzen Zahlen nach links `A << B`, rechts bei Beibehaltung des Vorzeichens `A >> B` und nach rechts ohne Beachtung des Vorzeichens `A >>> B`
8. Verknüpfung von Zeichenketten `A + B`
9. Summen `A + B`, `A - B`
10. Produkt, Quotient, Modulo `A * B`, `A / B`, `A % B`
11. Potenz `A ^^ B`
12. Negation von Zahlen `- A`, booleschen Ausdrücken `! A` und bitweises negieren von Zahlen `~ A`
13. geklammerter Ausdruck `( A )`, Variable und Literal

Dabei können `==`, `!=` und `.` Ausdrücke beliebigen Typs verarbeiten. Hingegen können `||`, `^`, `&&` und `!` nur boolesche und `<`, `<=`, `>`, `>=`, `+`, `-`, `*` und `/` nur arithmetische Ausdrücke und `%`, `<<`, `>>`, `>>>` und `~` ausschließlich ganze Zahlen verarbeiten. Mit `&` und `|` können boolsche Werte verknüpft werden und ganze Zahlen bitweise verbunden werden.

Geklammerte Ausdrücke werden von innen nach außen, Operationen gleicher Bindung von links nach rechts ausgewertet. Bei Arrays ist darauf zu achten, dass ein Ausdruck sie nicht als Ganzes verarbeiten kann. Stattdessen kann der Ausdruck nur einzelne Elemente des Arrays verarbeiten.

Ein Literal ist eine Zeichenkette oder eine ganze Zahl. Darüber hinaus ist es auch möglich, `null` als Platzhalter für „kein Knoten“ und die booleschen Literale `true` und `false` zu verwenden. Zusätzliche Literale für Zahlen sind `MAX_INTEGER` und `MIN_INTEGER`, die die größte beziehungsweise kleine speicherbare Zahl angeben. Es existieren darüber hinaus die Fließkommaliterale `PI` (3,14...), `E` (2,71...) und `NaN` (*not a number*).

Die Bindung von Gleichheits- und Vergleichsoperatoren ist dabei so gewählt, wie es auch bei Java üblich ist. Beispielsweise wird der Ausdruck `4<5 == 6>7` ausgewertet wie `(4<5)==(6>7)`.

Bei einem arithmetischen Fehler wird die betreffende Methode mit einer Fehlermeldung auf der Konsole beendet. Arithmetische Fehler sind beispielsweise Division durch 0 (`n/0`) oder Modulo 0 (`n%0`). Das Beenden betrifft nur die Methode, in der der Fehler aufgetreten ist. Insbesondere bedeutet das für eine Methode `a`, die eine Methode `b` aufruft, weiter ausgeführt wird. Dafür ist unerheblich, ob `b` durch einen Fehler terminiert oder ob `a` und `b` verschiedene Methoden sind.

2.6 Schleifen und Verzweigungen

Es gibt in NetSimLan zwei Arten von Schleifen `for` und `while`. Dazu existiert die Verzweigung `if-else`. Der Aufbau derartiger Elemente ist eine folgender Varianten:

```
1 for (Ass;Expr;Ass) Stmt
2 for (Type Ident : Expr) Stmt /* foreach array version */
3 for (Type IdentKey : Type IdentValue : Expr) Stmt /* foreach hashtabelle
     version */
4 while(Expr) Stmt
5 if (Expr) Stmt
6 if (Expr) Stmt else Stmt
```

`Ass` ist hierbei eine Zuweisung zu einer Variable. Diese ist optional und wird entgegen der üblichen Schreibweise nicht mit einem weiteren Semikolon beendet. Die erste Zuweisung wird vor dem Betreten der Schleife, die zweite nach jedem Schleifendurchlauf ausgeführt. Es ist hierbei jedoch nicht möglich, eine neue Variable einzuführen. Alle Variablen in `Ass` müssen bereits in dem Block definiert sein, in dem die Schleife liegt.
`Expr` ist (außer bei den foreach-Schleifen) ein boolescher Ausdruck. Die Schleife wird durchlaufen, solange der Ausdruck erfüllt ist. Bei der Verzweigung wird der erste Befehl `Stmt` ausgeführt, falls der Ausdruck erfüllt ist. Falls dies nicht der Fall ist, wird, falls vorhanden, der zweite Befehl `Stmt` nach `else` ausgeführt.
`Stmt` ist der Befehl, welcher in einem Schleifendurchlauf, beziehungsweise der Verzweigung, ausgeführt wird. Hier kann auch ein Block genutzt werden.
Die zweite und Schleife iteriert durch alle Elemente des Arrays in `Expr`. Dabei wird der Variable `Ident` der Wert des arrays bzw. `IdentKey` und `IdentValue` die Werte des Keys und Values der Hasttabelle zugewiesen. Diese Variable/Variablen ist/sind vom Typ `Type` und an dieser Stelle deklariert.
Eine Schleife lässt sich vorzeitig mit dem Befehl `break;` beenden. Ein Schleifendurchlauf lässt sich mit dem Befehl `continue;` überspringen.

3 Die grafische Oberfläche

Falls das zu simulierende Programm übersetzbar ist, erscheint nach dem Aufruf von NetSimLan eine grafische Oberfläche. Diese hat ein Fenster zur Darstellung der aktuellen Verbindungen zwischen den Knoten. Beim Start ist es ein leeres, weißes Fenster. Daneben öffnet sich das Hauptfenster. In dem Hauptfenster kann der Anwender die Simulation steuern, die einzelnen Knoten überwachen und die Textausgabe sehen. Dieses Fenster gliedert sich in drei in der Größe veränderlichen Bereiche. Abbildung 3 zeigt das Hauptfenster beispielhaft während der Simulation einer zweifach verketteten Liste mit Kreiskante und 30 Knoten.

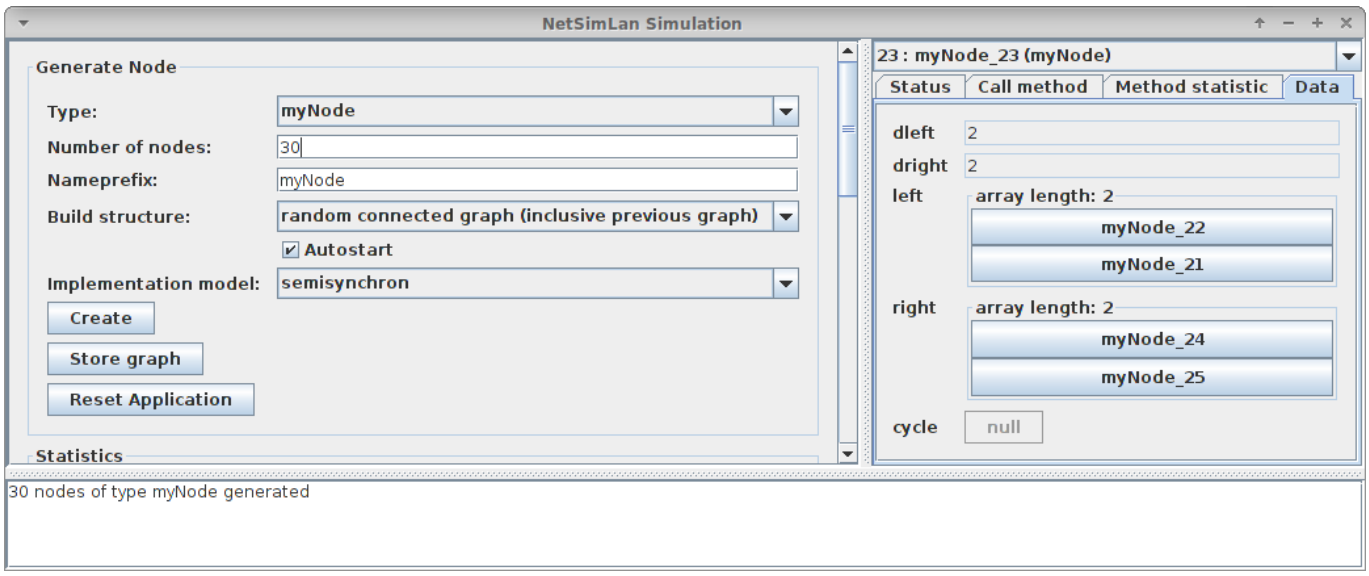


Abbildung 3: Das Hauptfenster nach dem Start einer Simulation.

3.1 Globaler Teil des Hauptfensters

Das Hauptfenster zeigt im linken oberen Bereich die globale Konfiguration. Hier lassen sich neue Knoten für die Simulation erstellen, Statistiken sehen, die Frequenz der *Timeouts* einstellen und die Visualisierung im anderen Fenster anpassen.

Knoten erstellen

Zum Erstellen eines oder mehrerer Knoten muss der Anwender zunächst den Typ angeben. Dies sind die Typen, welche im Programm definiert wurden. Wenn hier die Auswahl gewechselt wird, erscheint der Typ auch gleich als Vorschlag im Namensfeld. Der Vorteil ist, dass hierbei die Namen direkt Rückschlüsse auf den Knotentyp liefern und so das Erkennen im Graphen einfacher wird, falls mehrere Typen beteiligt sind. Falls der Anwender dies wünscht, kann er den Namen der neuen Knoten im Namensfeld ändern. Der tatsächliche Name setzt sich aus dem Präfix, der im Namensfeld angegeben wird und der fortlaufenden Identifikationsnummer des Knotens zusammen. Der Nutzer gibt außerdem an, wie viele Knoten erstellt werden sollen.

Unter *Build structure* legt der Anwender fest, in welcher Art die neuen Knoten untereinander verbunden werden. Es ist möglich, alle Knoten jeweils mit einem zufälligen anderen Knoten zu verbinden. Dieser Fall bildet hier eine Ausnahme, da sie nicht nur zufällig untereinander, sondern teilweise auch mit bereits vorher erstellten Knoten verbunden werden. Die zweite Möglichkeit ist, alle neuen Knoten mit einem bereits bestehenden Knoten zu verbinden. Hierbei wählt der Nutzer, ob ein zufälliger oder kein Knoten gewählt wird. Diese Liste enthält ebenfalls alle bereits erstellten Knoten, sodass sie danach separat zur Wahl stehen. Falls in dieser Konfiguration die Wahl auf *none* fällt, haben die neu erstellten Knoten keine Verbindung zu einem anderen Knoten. Die Wahl eines zufälligen Knotens weist dieses Verhalten auf, falls es sich bei den neuen Knoten um die ersten Knoten in der Simulation handelt. Begründet ist dieses Verhalten dadurch, dass ein Knoten aus der Menge jener Knoten ausgewählt wird, die vor diesem Schritt bereits bestehen. Diese Menge ist jedoch leer, falls noch kein Knoten erstellt wurde. Als weitere Struktur innerhalb der neu erstellten Knoten kann der Anwender eine Clique wählen. Hier muss der Nutzer angeben, ob alle neuen Knoten mit einem bestimmten, zufälligen oder keinem Knoten der bisherigen Struktur verbunden werden sollen. Weitere Strukturen sind eine lineare aufsteigende, eine lineare absteigende Verkettung und ein Gitter. Hierbei gibt der Anwender jeweils an, ob und mit welchem Knoten der erste Knoten dieser neuen Struktur verbunden werden soll. Die anderen Knoten verbinden sich hier initial nur innerhalb der Struktur.

Unter der beschriebenen Auswahl für die Struktur gibt der Nutzer an, ob die neu erstellten Knoten direkt gestartet werden sollen. Diese Auswahl ist beispielsweise sinnvoll, wenn der Anwender verschiedene Strukturen erstellen und gemeinsam starten möchte. Für jeden Knoten kann hier festgelegt werden, ob er semisynchron oder asynchron laufen soll. Es ist möglich den aktuellen Knoten zu speichern um ihn für Künftige Simulationen weiterzunutzen. Dies erläutere ich in Kapitel 5 genauer. Ferner kann der Nutzer hier die gesamte Simulation zurücksetzen. Dies ist gleichbedeutend mit einem Neustart der Anwendung. Allerdings wird hierbei der Quelltext des simulierten Programmes nicht neu übersetzt, was bei einem erneuten Aufruf von NetSimLan der Fall ist. Möchte der Anwender Änderungen an seinem Programm durchführen, muss er die Anwendung daher neu aus der Konsole starten.

Statistik

Unter der Knotengenerierung befindet sich die Statistik. Diese zeigt an, wie viele Knoten sich in welchem Status und wie viele Nachrichten sich maximal und durchschnittlich in der Eingangsschlange der Knoten befinden. Ebenfalls kann der Nutzer hier alle Knoten starten, die noch nicht gestartet sind. Auch ist es möglich, alle Knoten schlafen zu legen. Diese beiden Möglichkeiten sind deaktiviert, falls alle Knoten gestartet sind, beziehungsweise schlafen. Die Statistik wird automatisch einmal pro Sekunde aktualisiert. Am Ende der Statistik befinden sich zwei weitere Knöpfe. Mit dem ist es möglich, die Simulation zu pausieren und fortzusetzen. So kann der Nutzer eine Situation im Netzwerk genauer betrachten, wenn er das wünscht. Technisch gesehen werden beim Pausieren die Knoten daran gehindert, ihre Nachrichtenschlange abzuarbeiten und erhalten keine neuen Timeouts mehr, bis die Simulation fortgesetzt wird. Über den letzten Knopf lässt sich einsehen, wie häufig die einzelnen Methoden aufgerufen werden. Es ist als Gesamtsumme und als Durchschnitt pro Knoten angegeben.

Timeouts

Unter den Statistiken lässt sich einstellen, wie lang ein Timeout in den Knoten sein soll. Dadurch lässt sich die Simulation verlangsamen oder beschleunigen. Es sind 10 Millisekunden bis 10 Sekunden möglich, wobei der Standard 100 Millisekunden sind. Eine Änderung an dieser Stelle tritt in jedem Knoten beim nächsten Timeout in Kraft. Das kann dazu führen, dass eine Änderung auf einen niedrigen Wert verzögert sichtbar wird. Da auch in der Realität Computer nicht immer synchron zu anderen laufen, wird hier automatisch eine Unschärfe von $\pm 10\%$ eingebracht, um die jeder Timeout in jedem Knoten zufällig variiert. Dieses Vorgehen soll die Situation vermeiden, dass die Knoten sich verhalten, als wenn sie in Runden statt voneinander unabhängig arbeiten. Die Systemlast für das Simulationssystem hängt naturgemäß auch von dem eingestellten Timeout ab, sodass mitunter kleine Intervalle nicht richtig simuliert werden können.

Visualisierung und Ausgabe

Als letzten Punkt des globalen Bereiches ist es dem Nutzer möglich, die Visualisierung anzupassen. Er kann die Knotennamen in der Visualisierung ein- und ausblenden sowie die Kanten farblich entsättigen und dies rückgängig machen. Diese beiden Einstellungen können situationsabhängig die Übersicht in der Visualisierung erhöhen. Für den Fall, dass die impliziten Kanten in einer Simulation erwünscht oder unerwünscht sind, erlaubt ein dritter Knopf, die impliziten Kanten ein- beziehungsweise auszublenden. Weil ohne die Darstellung impliziter Kanten die Visualisierung seltener aktualisiert werden muss, wird die Visualisierung mit dieser Einstellung flüssiger dargestellt. Insbesondere bei einer großen Anzahl an Knoten macht sich das bemerkbar. Beim Programmstart sind die impliziten Kanten aus diesen Gründen ausgeblendet. Zur besseren Betrachtung des Graphen kann auch die automatische Positionierung aus- bzw. eingeschaltet werden. Zu Beginn werden Knoten automatisch positioniert. Als letzte Option der Nutzer die automatische Positionierung der Knoten in der Visualisierung deaktivieren beziehungsweise aktivieren.

Im unteren Bereich des Hauptfensters befindet sich die Ausgabe. Hier wird das Erstellen der Knoten mit Typ und Anzahl, die Ausgabe von `print`, Methoden mit Parametern durch den Nutzer und das Aufwachen der Knoten aus dem Status *schlafend* protokolliert.

3.2 Knotenspezifischer Teil des Hauptfensters

Nach dem Erstellen mindestens eines Knotens ist es dem Anwender möglich, im rechten oberen Teil des Hauptfensters jeweils einen Knoten zeitgleich zu überwachen und zu steuern. Der Bereich bietet oben eine Auswahl der Knoten. In dieser ist die ID, der Name und die Art der Knotens angegeben. Diese Auswahl heißt im weiteren Knotenauswahlmenü. In Abb. 3 steht in dieser Auswahl „`23 : myNode_23 (myNode)`“. Das bedeutet, dass der Knoten mit der ID `23`, dem Namen `myNode_23` und dem Typ `myNode` ausgewählt ist. Alle weiteren Funktionen im knotenspezifischen Bereich des Fensters beziehen sich auf den im Knotenauswahlmenü gewählten Knoten.

Unter dieser Auswahl befinden sich vier Reiter: Im Ersten ist der Status des Knotens zu sehen und es ist möglich, diesen zu verändern. Der Anwender kann den Knoten, sofern noch nicht geschehen, starten oder in Schlaf versetzen oder beenden. Dadurch kann der Nutzer beispielsweise einen Hardwaredefekt simulieren. Zusätzlich zeigt dieser Reiter, wie groß der Puffer für eingehende Nachrichten maximal und durchschnittlich ist und wie häufig bei dem Knoten ein Timeout ausgelöst wurde. Im zweiten Reiter lassen sich die Methoden des gewählten Knotens aufrufen. Der Anwender wählt die aufzurufende Methode und legt die Parameter fest. In der Auswahl entsprechen die Methoden jener Reihenfolge, in welcher sie im Quelltext beschrieben sind. Wenn der angezeigte Knoten zu einem anderen Knoten desselben Typs wechselt, bleiben diese Parameter erhalten. Der Wechsel erfolgt über das Knotenauswahlmenü, die Visualisierung (siehe 3.3) oder den Datenreiter. Letzterer wird später näher beschrieben. Bei einem Methodenwechsel oder Wechsel auf einen anderen Knotentyp werden die Parameter zurückgesetzt. Durch das Erhalten der Parameter ist es ohne hohen Aufwand möglich, denselben Aufruf bei verschiedenen Knoten durchzuführen. Gleichzeitig werden keine Ressourcen durch eine lange Speicherung von veralteten Daten belegt, da die Parameter nicht vorgehalten werden müssen.

Im dritten Reiter sind alle Methoden aufgelistet, die der Knoten implementiert hat. Hier ist einsehbar, welche Methode in welcher Anzahl aufgerufen wurde. Dabei zählen sowohl die Aufrufe von anderen Knoten, dem selben Knoten und dem Nutzer. Der vierte Reiter zeigt dem Nutzer die Attribute des aktuell dargestellten Knotens an. Dieser Reiter ist der in Abbildung 3 dargestellte Reiter. Diese Daten werden periodisch aktualisiert. In diesem Reiter können Daten nicht verändert werden. Zahlen und Zeichenketten kann der Anwender kopieren, falls er sie außerhalb der Simulation weiter verwerten möchte. Bei Attributen, die auf einen anderen Knoten zeigen, kann dieser andere Knoten durch einen Mausklick auf den Wert des Attributs ausgewählt werden. Hiermit ist es möglich, sich einfach in der entstanden Struktur zu bewegen. Diese Möglichkeit ist nicht gegeben, falls der Datenwert nicht gesetzt, also `null` ist. Arrays und Hashtabellen sind als Block dargestellt. Auch die Länge der Datenstruktur steht über dessen Daten, sodass der Anwender diese einfach einsehen kann. Wie so etwas aussehen kann, ist in Abbildung 3 an der Variable `left` zu sehen.

3.3 Visualisierungsfenster

Im zweiten Fenster wird der aktuelle Zustand des Verbindungsgraphen dargestellt. Die Knoten sind in der Visualisierung farblich kodiert. Der aktuell im Hauptfenster gewählte Knoten ist orange dargestellt. Die Anderen sind entsprechend ihres Zustandes gefärbt. Dabei steht schwarz für Knoten, die noch nicht gestartet wurden, blau für laufende, cyan für auf Synchronisierung wartende, gelb für schlafende und rot für tote Knoten. Knoten, die mit der Methode `mark` markiert werden, erhalten einen grauen Kreis um den Knoten. Standardmäßig werden explizite Kanten schwarz und implizite Kanten grün dargestellt. Das Ändern

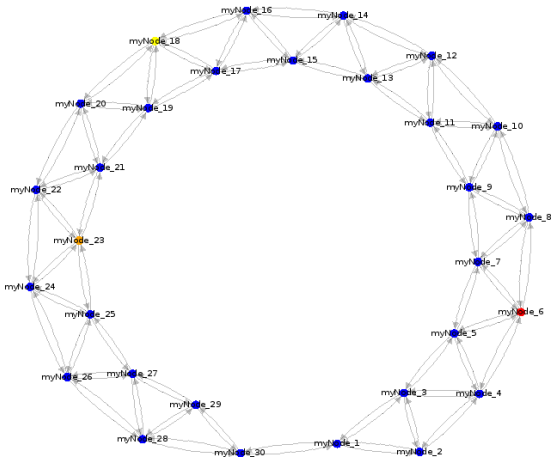


Abbildung 4: Die Visualisierung einer zweifach verketteten Liste mit Kreiskante.

der Kantenfarben ist in Kapitel 3.1 beschrieben. Falls zwischen zwei Knoten mehrere Kanten existieren werden diese gebogen dargestellt um sie unterscheiden zu können. Verbindungen, die nur in einer Variable innerhalb einer Methode gehalten werden, existieren nur sehr kurz. Um die Performance der Knoten nicht zu beeinträchtigen, werden diese kurzlebige Art von Kanten nicht dargestellt. Knoten werden zur besseren Performance nicht direkt aktualisiert sondern gesammelt und alle Änderungen periodisch durchgeführt.

In diesem Fenster können die Knoten mit gedrückter rechter Maustaste bewegt werden. Durch einen Klick mit der linken Maustaste kann der Anwender einen Knoten auswählen. Dieser Weg ist eine Alternative zum in 3.2 beschriebenen Knotenauswahlmenü. Das Hauptfenster und die Visualisierung passen sich entsprechend an. Der Nutzer kann mit dem Mausrad die Ansicht und den Tasten „Bild↑“ beziehungsweise „Bild↓“ vergrößern und verkleinern. Den angezeigten Ausschnitt kann er mit den Pfeiltasten bewegen.

Abbildung 4 zeigt beispielhaft eine Visualisierung. Hier ist eine zweifach verkettete Liste mit Kreiskante dargestellt. Diese Abbildung stellt das Gegenstück zu dem Fenster in Abb. 3 dar. In Abbildung 4 ist `myNode_6` tot, `myNode_18` schläft und `myNode_23` ist aktuell ausgewählt. Außerdem sind die Kanten per Hauptfenster blass gefärbt und die impliziten Kanten ausgeblendet.

4 Synchronisierung

NetSimLan bietet semisynchrone und asynchrone Kommunikation. Jeder semisynchrone Knoten arbeitet bei jedem periodisch eintretenden Timeout zuerst alle Nachrichten ab, die ihn bis zu diesem Zeitpunkt erreicht haben. Danach führt der Knoten die Methode `timeout` aus, sofern diese implementiert ist. Asynchrone Knoten arbeiten entweder bei einem Timeout zufällig viele Nachrichten ab oder ruft die Methode `timeout` auf.

Neben diesen Kommunikationsmodi gibt es explizite Synchronisierung. Hierzu kann ein Knoten die Methode `synchronize()` aufrufen. Sobald alle (nicht toten) Knoten `synchronize()` aufgerufen haben, wird auf allen Knoten `startRound()` ausgeführt, sofern das implementiert ist. Zwischen dem eigenen und dem global letzten Aufruf von `synchronize()` befindet sich der jeweilige Knoten im Zustand „Warte auf Synchronisierung“ und wird in der Visualisierung in cyan dargestellt.

5 Datei für initialen Graph

Beim Start der Software kann der Nutzer als zweiten Parameter einen initialen Graphen angeben. Dieser ist in einer Textdatei gespeichert, die zeilenweise verarbeitet wird. Jede der hier beschriebenen Zeilen ist optional und wird ignoriert, wenn sie falsch geschrieben ist oder aus anderen Gründen nicht verarbeitet werden kann. Über die Schalter `autolayout on/off`, `showimplicit on/off` und `autostart on/off` kann der Nutzer die Visualisierung steuern und angeben, ob die Simulation direkt gestartet werden soll. Falls einer der Werte nicht angegeben ist, gelten die Standardwerte `autolayout on`, `showimplicit off` und `autostart off`. Die Angabe `autostart on` bezieht sich auf alle Knoten, unabhängig davon, wo in der Datei die Zeile steht.

Ein Knoten wird mit dem Schlüsselwort `new` eingeleitet. Danach muss die Angabe des Knotentyps und ein Knotenname folgen. Diese Angaben werden durch ein Leerzeichen getrennt. Die Knotennamen müssen eindeutig sein und werden nicht um ihre ID ergänzt. Hierin unterscheidet sich dieses Vorgehen von der Generierung über die GUI. Optional kann das Schlüsselwort `asynchron` oder `semisynchron` folgen, um die Kommunikationsweise des Knotens festzulegen. Ohne diese Angabe wird der Knoten semisynchron erstellt. Wenn alle diese Parameter angeben sind kann mit zwei weitere Parametern die Position in der Visualisierung als Fließkommazahl angeben werden. Hierbei kann der Nutzer den Maßstab des Koordinatensystems frei wählen. Die Visualisierung passt sich entsprechend an. Wenn die automatische Anordnung ausgeschaltet ist, bietet es sich an, alle diese Werte anzugeben.

Kanten werden durch die Namen der zwei beteiligten Knoten verbunden mit einem Pfeil `->` erstellt. Für die Knoten `a` und `b` sieht die Kante folgendermaßen aus: `a -> b`. Alle Zeilen können beliebig in der Datei angeordnet werden. Die einzige Einschränkung hierbei ist, dass für Kanten nur Knoten zur Verfügung stehen, die in Zeilen darüber angelegt wurden.

Ein kompletter Graph kann für das Beispiel `broadcast` folgendermaßen aussehen:

```
1 autolayout on
2 showimplicit off
3 autostart on
4 new Server server
5 new Client client1
6 new Client client2
7 server -> client1
8 server -> client2
```