Monitoring Server für ein P2P-basiertes Live-Streaming-System

Monitoring Server for a P2P-based Live-Streaming-System
Bachelor-Thesis von Thorsten Jacobi
März 2012

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Peer-to-Peer Netzwerke und

Monitoring Server für ein P2P-basiertes Live-Streaming-System
Monitoring Server for a P2P-based Live-Streaming-System

Vorgelegte Bachelor-Thesis von Thorsten Jacobi

Prüfer: Professor Dr. Thorsten Strufe
Verantwortlicher Mitarbeiter: Benjamin Schiller

Tag der Einreichung:

Inhaltsverzeichnis

Zusammenfassung

Zur Zeit erfreuen sich Live-Streaming-Systeme einer immer höheren Beliebtheit. Da aber die klassische Client-Server-Architektur durch die steigende Anzahl von Nutzern nicht gut skaliert, wird oft auf eine Peer-to-Peer basierte Lösung zurückgegriffen. Daher wurde auch an der TU Darmstadt ein solches System entwickelt. Allerdings ist dieses, wie auch andere auf Peer-to-Peer basierten Systeme, wegen seiner Komplexität nicht leicht zu analysieren. Deswegen wurde im Rahmen dieser Arbeit ein Monitoring Server für dieses existierende Peer-to-Peer-Streaming-System entwickelt. Dieser Monitoring Server stellt das Netzwerk grafisch da und kann in dieses auch aktiv eingreifen. Dadurch kann das Netzwerk besser analysiert und getestet werden.

1 Einleitung

Zur Zeit wird Live-Streaming immer beliebter und die Qualitätsansprüche der Zuschauer erhöhen sich. Durch den Zuwachs an Zuschauern sowie deren Ansprüche an die Qualität wird immer mehr Bandbreite benötigt, welche mit einer klassichen Client-Server-Architektur nur sehr kostenintensiv zu realisieren ist. Eine Alternative zu diesem Ansatz ist die Ausnutzung der Bandbreite der einzelnen Peers und somit die Umstellung der Architektur auf ein Peer-to-Peer(P2P) Netzwerk. Allerdings bringt diese Umstellung auch Probleme mit sich. So können in einem P2P Netzwerk jederzeit neue Peers hinzukommen und verbundene Peers das Netzwerk verlassen. Auf Grund dieser Aktivitäten muss das Overlay von diesem Netzwerk mit Hilfe von Algorithmen permanent angepasst werden. Um diese Anpassungen besser analysieren sowie verschiedene Algorithmen testen zu können, wird eine grafische Visualisierung des Netzwerks mit der Möglichkeit aktiv in dieses einzugreifen benötigt. Daher wird in dieser Arbeit ein grafischer Monitoring Server für ein bestehendes P2P-Streaming-Netzwerk entwickelt. Dieser wird im weiterem Verlauf dieser Arbeit als Monitor bezeichnet.

Diese Arbeit ist folgendermaßen strukturiert. Zunächst werden im Kapitel 2 Grundlagen von Live-Streaming-Systemen vorgestellt. Anschließend wird in Kapitel 3 das existierende System sowie die geplante Erweiterung, der Monitor, vorgestellt. Im Abschnitt 4 werden die gestellten Anforderung an den Monitor aufgelistet. Im darauf folgenden Abschnitt 5 wird die geplante Realisierung der einzelnen Anforderungen erläutert. Der anschließende Abschnitt 6 befasst sich mit der Implementierung des Monitors. Schließlich wird in Abschnitt 7 die Verwendung des Monitors beschrieben und in Abschnitt 8 eine Zusammenfassung sowie einen kurzen Ausblick auf die Erweiterungsmöglichkeiten des Monitors gegeben.

2 Grundlagen: Live-Streaming-Systeme

In diesem Kapitel werden verschiedene Ansätze vorgestellt mit denen man ein Live-Streaming-System realisieren kann. Dabei werden zunächst die Unterschiede zwischen einer Client-Server-Architektur und einer Peer-to-Peer Architektur aufgezeigt. Anschließend wird der Unterschied zwischen Pull- und Push-basierten Systemen sowie verschiedene Overlays für eine P2P Architektur dargestellt.

2.1 Client-Server-Systeme

Die Client-Server-Systeme waren der Anfang des Live-Streamings. Bei diesen Systemen werden Server bereitgestellt zu denen sich alle alle Zuschauer bzw. Clients verbinden und den Stream von diesen gesendet bekommen. Durch die einfache Funktionsweise sind diese Systeme leicht zu implementieren. Allerdings haben diese Systeme ein großes Problem mit der Skalierbarkeit, da jeder Server nur eine beschränkte Anzahl von Clients bedienen kann muss die Anzahl der Server proportional zu der Anzahl der Clients steigen und hat somit eine schlechte Skalierbarkeit.

2.2 P2P-Systeme

Im Gegensatz zu Client-Server-Systemen, bei denen die Rollen klar verteilt sind, ist bei einem P2P-System jeder Peer sowohl Empfänger als auch Sender, d.h. jeder Peer empfängt den Stream von Peers und sendet die empfangenen Daten an andere Peers weiter. Dadurch kann die Bandbreite der Peers genutzt werden um weitere Peers zu versorgen. Allerdings haben diese Systeme nicht nur Vorteile. Ein großes Problem ist die Heterogenität der Peers, da viele Peers unterschiedliche Bandbreiten haben, kann nicht jeder Peer die gleiche Anzahl von anderen Peers versorgen. Durch die Heterogenität entsteht auch ein weiteres Problem für neue Peers die dem bestehenden Netzwerk beitreten möchten, da sie beim Beitritt zunächst ihre Position im Netzwerk finden, d.h. sie müssen andere Peers finden die ihnen den Stream zur Verfügung stellen. Da die Streams in Echtzeit übertragen werden, muss bei diesen Systemen auch auf die Verzögerung, die durch die Verteilung über die Peers entsteht, geachtet werden.

2.3 Push- und Pull-Ansatz

Ein wichtiger Aspekt bei diesem System ist die Verteilung der Daten. Hierfür gibt es die Möglichkeiten der *Push*- sowie der *Pull*-basierten Übertragung. Bei einem *Push*-basierten System registrieren sich die Peers bei einem Sender als Empfänger. Sobald der Sender neue Daten empfängt sendet er diese sofort an alle als Empfänger registrierten Peers weiter. Im Gegensatz dazu werden in einem *Pull*-basierten System nur dann neue Daten gesendet, wenn ein Peer diese selber anfragt. Da in einem *Push*-basierten System diese Anfragen nicht nötig sind, haben diese Systeme ein geringere Verzögerung und einen geringeren Overhead.

2.4 Mesh-basiertes Overlay

In jedem P2P-System muss definiert werden, wie die Peers sich miteinander verbinden bzw. welche Peers für Verbindungen ausgewählt werden. Die einfachste Möglichkeit hierbei ist ein Mesh-Netzwerk. Bei einem solchen Netzwerk gibt es keine feste Struktur, d.h. jeder Peer verbindet sich dynamisch mit anderen verfügbaren Peers. Dabei werden die Peers mit denen Verbindungen aufgebaut werden meist mit Hilfe von Metriken ausgewählt, zum Beispiel werden Peers zu denen eine kurze Latenz besteht bevorzugt. Ein solches Overlay wird in der Abbildung

1 dargestellt. Des Weiteren wird in einem Mesh-basierten System meist der Pull-Ansatz verwendet. Dieser Ansatz kombiniert mit den unbestimmten Verbindungen zwischen den Peers macht dieses System besonders robust, da bei einem Ausfall von einem Peer die Daten von belibigen anderen Peers angefragt werden können. Als Beispiele für ein solches System kann Collstreaming/DoNet [XLKZ07] sowie PPlive [HLR08] genannt werden.

Abbildung 1: Mesh-Overlay

2.5 Tree-basiertes Overlay

Eine Alternative zu den unstrukturierten *Mesh*-Netzwerken sind die Baum-basierten Netzwerke. Bei diesen Netzwerken hat jeder Peer einen festen Platz in der Baumstruktur und verbindet sich somit nur mit seinem Elternknoten sowie seinen Kindern. Da in diesen Systemen die Quelle die Wurzel des Baumes ist, bekommen die Peers die Daten immer nur von ihren Elternknoten. Auf Grund des Datenflusses wird für diese Systeme meist der *Push*-Ansatz verwendet. Durch die Nutzung des Push-Ansatz sowie der Baumstruktur wird eine sehr schnelle Weiterleitung der Daten gewährleistet, wodurch der Stream bei den Empfängern mit nur einer kurzen Verzögerung ankommt. Durch diese Verteilung der Daten sind diese Systeme aber nicht sehr robust, da bei einem Ausfall von einem Knoten der gesamte Teilbaum des Knotens von den Daten abgeschnitten ist. Ein weiteres Problem ist, dass die Peers an den Blättern des Baumes ihre Bandbreite nicht zur Verfügung stellen, da sie keine Kinder haben.

Um den Problemen des Baum-basierten Netzwerks entgegen zu wirken, wird meist ein *Multi-Tree* Overlay genutzt. Hierbei wird der Stream in mehrere Teilstreams, welche *Stripes* genannt werden, aufgeteilt. Dabei wird der Stream zunächst in einzelne Blöcke unterteilt, welche fortlaufend nummeriert werden. Danach wird jeder Block dem Stripe zugewiesen, der den Restwert seiner Nummer geteilt durch die Anzahl der Stripes entspricht. Ein Bespiel für eine solche Aufteilung wird in Abbildung 4 dargestellt. Anschließend wird für jeden Stripe eine eigene Baumstruktur aufgebaut über die die Daten des Stripes verteilt werden. Hierbei bekommt jeder Peer in jedem Baum eine andere Position. Hierdurch wird gewährleistet, dass ein Peer nicht in jedem Baum ein Blattknoten ist und somit auch Daten weiterleiten kann. Des Weiteren hat es den Vorteil, dass bei einem Ausfall von einem Peer meist für die Kinder nur ein Stripe ausfällt. Diesen Ausfall von einem Stripe können moderne Codecs kompensieren, so dass der Stream immer noch angezeigt werden kann.

Die beiden Overlays werden in den Abbildungen 2 und 3 dargestellt. Hierbei kann als Beispiel für ein Single-Tree-basiertes System Overcast [JGJ+00] sowie für ein Multi-Tree-basiertes System das in dieser Arbeit verwendete System von Strufe [Str07] gennannt werden.

Abbildung 2: Single-Tree-Overlay

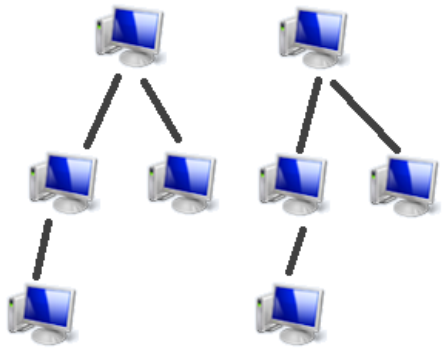

Abbildung 3: Multi-Tree-Overlay

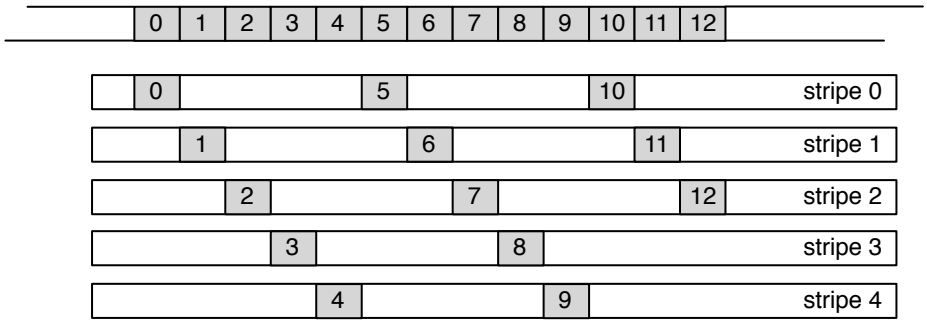

Abbildung 4: Aufteilung eines Streams in fünf Stripes [ADS+11]

3 Existierendes System und geplante Erweiterung

In diesem Kapitel wird das bereits existierende P2P-basierte Live-Streaming-System sowie die geplante Erweiterung vorgestellt. Dazu wird zunächst einige allgemeine Informationen über das System aufgezeigt. Anschließend werden die einzelnen Komponenten des Systems sowie abschließend die geplante Erweiterung, der Monitor, vorgestellt.

Das existierende System ist eine Implementierung des von Strufe [Str07] vorgestelltem Systems. Es ist ein Multi-Tree-basiertes System. Bei dem die Peers die Baumstrukturen nur mit den Informationen über ihre Kinder sowie Eltern optimieren. Es besteht aus den Komponenten *Tracker*, *Source* und *Peer* und soll nun um einen Monitor erweitert werden. Der Nachrichtenverkehr zwischen den bestehenden Komponenten ist in Abbildung 5 dargestellt.

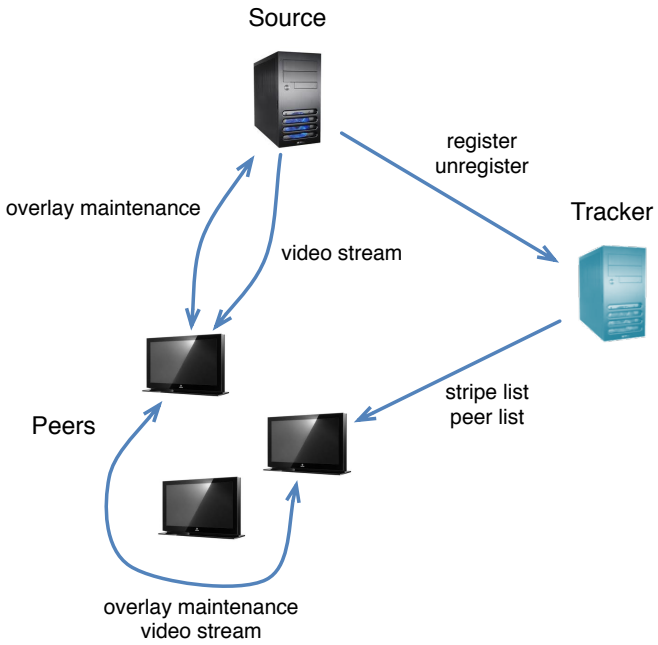

Abbildung 5: Nachrichtenverkehr zwischen den Komponenten [ADS+11]

3.1 Tracker

Der *Tracker* ist die zentrale Komponente des Netzwerks. Seine Aufgaben bestehen aus dem Verwalten von Informationen. Zu diesen Informationen zählt eine Übersicht von allen verfügbaren Streams sowie der jeweiligen Source und den Peers, die diesen Stream schauen. Die Stream-Informationen bekommt er, da die Source die Streams bei ihm registriert und beim Beenden wieder abmeldet. Außerdem ist er für die Herausgabe einer initialen Peerliste an neue Peers zuständig. Diese Peerliste benötigen die Peers um einen Einstieg in das Netzwerk zu finden. Die Informationen der Peers erhält er dadurch, dass die Peers bei ihm nach den Streams sowie nach einer initialen Peerliste für den gewählten Stream anfragen.

3.2 Source

Die *Source* ist der Ursprung des Streams. Ihre Aufgabe ist den Stream in mehrere Stripes aufzuteilen und die Daten der Stripes an die verbundenen Peers weiter zu verteilen. Dadurch ist sie die Wurzel von jedem Baum des Overlays. Dabei versucht sie auch die Verbindungen mit

ihren Kindern zu optimieren und verhält sich hauptsächlich wie ein normaler Peer. Außerdem registriert sie beim Beginnen des Streamings den Stream beim Tracker und meldet diesen wieder ab, wenn sie das Streaming beendet.

3.3 Peer

Der Peer ist ein Zuschauer des Streams. Er verbindet sich zunächst mit dem Tracker um eine Liste der Streams zu bekommen. Wenn er sich für einen Stream entschieden hat, fragt er beim Tracker eine initiale Peerliste an. Anschließend verbindet er sich für jeden Stripe mit einem Peer aus der Liste. Sobald er sich mit diesen verbunden hat, positioniert er sich innerhalb der jeweiligen Baumstruktur. Nachdem er seine Position gefunden hat, beginnt er damit die Daten des Streams zu empfangen. Sobald neue Peers sich mit ihm verbinden und seine Kinder werden sendet er die Daten, die er empfängt, an diese weiter. Des Weiteren beobachtet er während des Streaming seine verbundenen Peers und versucht das Overlay gegebenenfalls zu optimieren. Dabei hat er die Möglichkeit eigene Kinder an andere Kinder weiterzuleiten. Dies wird zum Beispiel genutzt, wenn er nicht genügend Bandbreite für alle Kinder zur Verfügung hat. Außerdem hat er auch die Möglichkeit die Kinder von seinen Kinder zu seinen eigenen Kindern zu machen. Dies kann er nutze, wenn zum Beispiel mehr Bandbreite verfügbar hat und somit mehr Kinder mit Daten versorgen könnte. Durch das Hinaufziehen von Kindern wird die Verzögerung des Streams minimiert. Des Weiteren fügt er die Daten der einzelnen Stripes wieder zu dem Stream zusammen um diesen auch darstellen zu können.

3.4 Erweiterung: Monitor

Zu den bestehen Komponenten soll nun ein Monitor implementiert werden. Dieser soll im Stande seien das Netzwerk zu visualisieren. Hierfür benötigt er zunächst Informationen über die verfügbaren Streams, die Source von diesem Stream sowie die Peers die diesen Stream schauen. Allerdings kann mit diesen Informationen die Struktur des Streams noch nicht dargestellt werden, da die Verbindungen zwischen den Peers noch nicht bekannt sind. Also müssen anschließend die Peers sowie die Source kontaktiert werden und diese nach Informationen über ihre Verbindungen befragt werden. Diese Verbindungen sind in Abbildung 6 dargestellt. Mit den nun vorhandenen Informationen kann das Netzwerk vollständig angezeigt werden.

Außerdem soll er in der Lage sein die Peers zu manipulieren. Dazu zählt die Anpassung von Eigenschaften der Peers sowie das Beenden dieser.

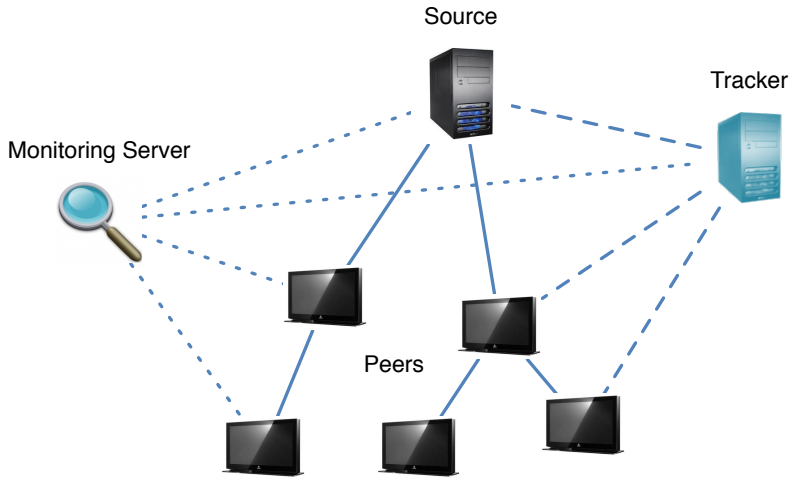

Abbildung 6: Verbindung zwischen den Komponenten mit dem Monitor [ADS+11]

4 Anforderungen

Dieser Abschnitt gibt einen Überblick über die gestellten Anforderungen welche bei der Entwicklung des Monitors berücksichtigt werden mussten.

Die Anforderungen können entsprechen der Ziele des Monitors, wie in Abschnitt 3.4 dargestellt, in die folgenden Aspekte eingeteilt werden.

4.1 Visualisierung der Informationen

Hierbei ist die übersichtliche Darstellung ein wichtiger Aspekt. Daher soll immer ein vollständiger Stream angezeigt werden, d.h. es soll die Baum-Struktur von allen Stripes gleichzeitig erkennbar sein. Des Weiteren soll ersichtlich sein welche Peers auf welcher Ebene der Baumstruktur liegen und Peers, welche nicht mit dem Hauptbaum des Stripes verbunden sind, sollen getrennt dargestellt werden. Außerdem sollen die Eigenschaften der Peers, wie zum Beispiel die Bandbreite, dargestellt werden.

4.2 Sammlung von Informationen

Hierzu zählt eine Übersicht aller im Netzwerk verfügbaren Streams sowie deren Peers und die Verbindungen zwischen den Peers. Daher muss der Monitor in der Lage sein mit den anderen Komponenten des Netzwerks zu kommunizieren, d.h. er muss Nachrichten senden und empfangen können. Außerdem muss er die gesammelten Informationen in einer geeigneten Datenstruktur speichern .

4.3 Manipulation des Netzwerks

Hierbei soll es möglich sein den Peers zu befehlen das Netzwerk geregelt zu verlassen. Hierbei meldet sich der Peers beim Tracker ab und leitet seine Kinder an andere Peers weiter so dass diese den Stream ohne Unterbrechung empfangen können. Im Gegensatz dazu soll es auch möglich sein den Peers zu befehlen einen Absturz zu simulieren. Dabei verlassen die Peers das Netzwerk sofort ohne sich vorher beim Tracker und bei den verbundenen Peers abzumelden. Durch diese Befehle können verschiedene Szenarien getestet werden und somit das Verhalten der Peers beurteilt werden. Des Weiteren soll es möglich sein die Eigenschaften

der Peers anzupassen. Zum Beispiel könnte durch die Anpassung der Bandbreiten-Eigenschaft das Verhalten des Peers bei zu wenig oder zu viel Bandbreite getestet werden.

4.4 Konfigurationsmöglichkeiten

Bei der Visualisierung soll es möglich seien die Anordnung der Stripes sowie der Peers innerhalb der Stripes anzupassen. Außerdem soll die Möglichkeit bestehen für verschiedene Peer-Typen unterschiedliche Bilder zu hinterlegen welche bei der Visualisierung der Peers anstatt des Standardbildes genommen werden. Des Weiteren soll es die Möglichkeit geben die Art der Eigenschaften des Peers in einer Konfigurationsdatei zu spezifizieren. Dabei soll angegeben werden mit welcher Bezeichnung die Eigenschaft in der Benutzeroberfläche dargestellt wird sowie welche Werte sie annehmen kann. Hier kann zum Beispiel die Bandbreite auf den Typ Zahl festgelegt werden sowie ein minimal und ein maximal Wert festgelegt werden.

5 Realisierung der Anforderungen

Im Folgenden werden die Ansätze zur Realisierung der in Abschnitt 4 gestellten Anforderungen vorgestellt.

5.1 Visualisierung der Informationen

Die Visualisierung der Netzwerkstruktur wird mit Hilfe eines Graphen-Frameworks realisiert. Dabei wird immer ein Stream mit allen Stripes und Peers angezeigt. Hierbei gibt es die Möglichkeit für die Anordnung der Stripes sowie der Peers verschiedene Layouts zu nutzen. Bei der Anordnung der Stripes kann beeinflusst werden wie viele Stripes jeweils in einer Zeile angezeigt werden. Bei der Anordnung der Peers gibt es die Auswahl zwischen einer festen Baum-basierten Anordnung und einer Ebenen-basierten Anordnung. Bei der Ebenen-basierten Anordnung werden die Kinder einer Ebene auf die gesamte Breite der darunter liegenden Ebene aufgeteilt. Hierdurch entsteht der Vorteil dass Peers bei einer hohen Anzahl von Ebenen mit wenigen Kindern übersichtlicher dargestellt werden, da der gesamte Platz ausgenutzt werden kann und so keine Peers zu dicht nebeneinander dargestellt werden müssen. Des Weiteren wird eine Möglichkeit geschaffen mit der die Layout-Einstellung während der Laufzeit angepasst werden kann.

5.2 Sammlung von Informationen

Zunächst benötigt der Monitor eine Liste der verfügbaren Streams. Hierzu sendet er die Nachricht *MGetStreams* an den Tracker. Die darauf folgende Antwort enthält eine Liste von allen registrierten Streams. Hierbei wird für jeden Stream ein *Identifier*, die Anzahl der Stripes sowie die Adresse der Quelle übermittelt. Der *Identifier* dient zur eindeutigen Identifikation des Streams. Nachdem der Monitor diese Informationen bekommen hat, benötigt er eine Übersicht der Peers für den jeweiligen Stream. Da sich die Peers bei dem Tracker registrieren bekommt er diese Informationen mit der Nachricht *MGetAllPeersForStream*. Diese Nachricht benötigt den Stream-*Identifier* als Parameter damit der Tracker weiß welche er Peers er senden muss. Als Antwort wird eine Liste mit den Adressen von allen bekannten Peers gesendet.

Mit den nun vorhandenen Informationen kann noch keine Struktur erstellt werden, da die Verbindungen zwischen den Peers noch nicht bekannt sind. Um die Struktur zu ermitteln, werden die Nachrichten *GetChildren* und *GetParents* genutzt. Der Monitor sendet diese direkt an die Peers und im Fall von *GetChildren* ebenfalls an die Source. Hierbei liefert *GetChildren* eine Liste von allen verbundenen Kindern. In dieser Liste wird für jedes Kind die Adresse und der Stripe angegeben. Die Nachricht *GetParents* liefert eine Liste mit den Eltern von denen der Peer die Daten bekommt. Diese Liste enthält für jeden Vater die Adresse sowie die Nummer des Stripes. Mit diesen Informationen kann eine vollständige Struktur der Bäume erstellt werden. Die Nachrichten *GetChildren* und *GetParents* sind zwar in einem korrekt aufgebauten System redundant, diese können aber in anderen Fällen genutzt werden um die Korrektheit der Informationen zu prüfen und so Fehler anzuzeigen.

Da in den oben beschriebenen Schritten eine Kommunikation mit den anderen Komponenten des Systems benötigt wird, ist die ein sehr wichtiger Bestandteil des Monitors. In den anderen Komponenten des Systems wird bereits eine einheitliche Abstraktion für das Senden und Empfangen von Nachrichten verwendet, welche auch beim Monitor eingesetzt werden soll. Durch die Verwendung der gleichen Abstraktion wird die Kompatibilität zwischen den

Komponenten sichergesellt. Alle versendeten Nachrichten beinhalten neben dem Typ der Nachricht auch eine ID, die Sendezeit sowie vom Typ abhängige zusätzliche Informationen.

Die Daten des Netzwerks werden in den drei Klassen *Tracker*, *Stream* und *Peer* gespeichert. Die Klasse *Tracker* ist für die Speicherung der Adresse des Trackers sowie der Liste mit allen verfügbaren Streams zuständig. Die Objekte der Klasse *Stream* speichern die Adresse der Source, den Identifier, eine Liste mit allen Peers sowie für jeden Stripe einen Graphen. Diese Graphen werden genutzt um die Verbindungen zwischen den Peers in den einzelnen Stripes zu speichern. Die Klasse *Peer* speicher die Adresse des Peers. Die Klassen sind auch in der Abbildung 7 dargestellt. Außerdem werden die verwendeten Nachrichten in der Tabelle 1 zusammengefasst.

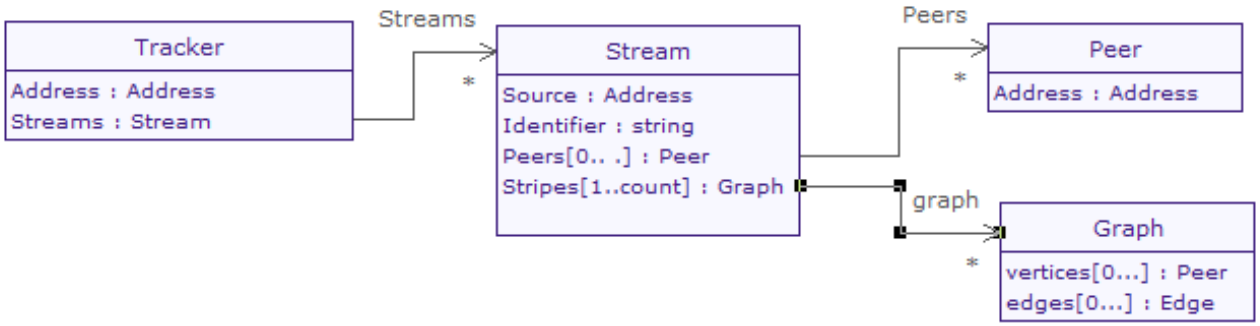

Abbildung 7: Entwurf der Datenstruktur

Bezeichnung	möglicher Empfänger	Parameter
MGetStreams	Tracker	-
MGetAllPeersForStream	Tracker	Stripe Identifier
GetParents	Peer	-
GetChildren	Peer und Source	-
GetInfo	Peer und Source	-
SetProperties	Peer und Source	Eigenschaften
Kill	Peer und Source	-
Term	Peer und Source	-

Tabelle 1: Liste der verwendeten Nachrichten

5.3 Manipulation des Netzwerks

Für die Manipulation des Netzwerks wird es eine Auflistung von allen ausgewählten Peers geben. Dabei werden auch die Eigenschaften der einzelnen Peers aufgelistet und können dort auch angepasst werden. Um diese Eigenschaften von einem Peer zu erfahren, sendet der Monitor die Nachricht *GetInfo* an den Peer. Dieser antwortet darauf mit einer Liste von Paaren. Diese Paare stellen den Eigenschaftsname sowie den Eigenschaftswert dar. Wenn diese Eigenschaften angepasst werden sollen, werden sie im Benutzerinterface geändert und anschließend mit der Nachricht *SetProperties* an den Peer gesendet. Bei der *SetProperties* Nachricht werden die Eigenschaften ebenfalls als eine Liste von Paaren übergeben.

Des Weiteren werden Schaltflächen eingefügt um einzelne Peers oder auch alle ausgewählten Peers zum Beenden zu zwingen. Hierfür gibt es die Nachricht *Term* welche den Peer dazu zwingt das Netzwerk ordnungsgemäß zu verlassen und die Nachricht *Kill* welche den Peer zu einem spontanen Verlassen des Netzwerks zwingt. Mit diesen beiden Befehle können verschiedene Verhaltensweisen der anderen Peers getestet werden.

Die verwendeten Nachrichten sind in der Tabelle 1 aufgelistet.

5.4 Konfigurationsmöglichkeiten

Die Möglichkeit der Konfiguration des Monitors wird mit Hilfe einer Konfigurationsdatei realisiert. Diese wird im XML-Format gespeichert. Sie bietet die Möglichkeit für jeden Peertyp ein Bild zu hinterlegen welches bei der Visualisierung im Graphen genutzt wird. Des Weiteren bietet sie die Möglichkeit Layouts für die Anordnung der Stripes zu definieren und bekannte Eigenschaften von Peers zu definieren so dass ein Eigenschaftstyp und mögliche Werte festgelegt werden können sowie eine Bezeichnung für die Anzeige im Benutzerinterface.

6 Implementierung

In diesem Kapitel wird die Implementierung des Monitors vorgestellt. Dabei wird zunächst auf die verwendeten Bilbiotheken eingegangen. Anschließend wird die Implementierung mit Hilfe der Aspekte *Netzwerkkommunikation*, *Graphen-Framework*, *Konfiguration*, *Benutzerinterface* vorgestellt.

6.1 Verwendete Bibliotheken

Für die Darstellung der Graphen wurde das Framework JUNG [1] ausgewählt, da es alle notwendigen Tools bietet und sehr flexibel ist. Es enthält direkt Klassen um Baum-basierte Graphen speichern zu können welche der Monitor für die Speicherung der Baumstruktur der Peers benötigt. Außerdem bietet es eine komplette Visualisierungs-Engine für die Darstellung von Graphen an. Diese ist sehr flexibel so dass eigene Algorithmen für das Layout des Knoten geschrieben werden können sowie die gesamte Darstellung des Graphen geändert werden kann. Hierzu zählt zum Beispiel die Form und Farbe der Knoten, die Beschriftung der Knoten sowie die Möglichkeit Knoten mit Bildern darzustellen. Außerdem können für verschiedene Ereignisse, wie zum Beispiel das Klicken auf einen Knoten, Methoden hinterlegt werden.

6.2 Netzwerkkommunikation

Für die Kommunikation mit den anderen Komponenten des Systems ist die Klasse *lso.monitor.Network* zuständig. Sie nutzt die Klasse *lso.MaintenanceNode* zum Senden und Empfangen von Nachrichten. Diese Klasse stellt die Methode *send(Address, MaintenanceMessage)* zur Verfügung welche zum Senden von Nachrichten verwendet wird. Als Parameter erwartet sie die Adresse der Komponente an die gesendet werden soll sowie die Nachricht. Sie arbeitet synchron, d.h. sie blockiert die weitere Ausführung bis sie eine Antwort erhalten hat bzw. bis ein interner Timer abgelaufen ist. Anschließend gibt sie die Antwort als *MaintenanceResponse* zurück. Die Nachrichten werden mit der Abstraktion über Google ProtocolBuffers[2] definiert und automatisch als Erben der Klasse *MaintenanceMessage* erzeugt. Analog dazu werden auch die Antworten automatisch als Erben der Klasse *MaintenanceResponse* erzeugt. Diese Struktur wird auch in Abbildung 8 dargestellt.

Wie bereits erwähnt stellt die Klasse *Network* sämtliche Methoden für die Kommunikation mit den anderen Komponenten zur Verfügung. So bietet sie für die Abfrage der verfügbaren Streams die Methode *updateStreams(Tracker)* an. Diese Methode bekommt den Tracker übergeben von dem die Streams aktualisiert werden sollen. Zum Aktualisieren sendet sie die Nachricht *MGetStreams* an den Tracker und bekommt als Antwort ein Objekt der Klasse *MStreams* welches alle beim Tracker registrierten Streams beinhaltet. Anschließend werden die neuen Streams dem Tracker hinzugefügt und nicht mehr vorhandene Streams gelöscht.

Zum Aktualisieren der Peers sowie der Verbindungen zwischen den Peers wird die Methode *updateStreamStructure(Tracker, Stream)* bereit gestellt. Diese benötigt den Stream der aktualisiert werden soll sowie der Tracker auf dem der Stream registriert ist. Sie sendet zunächst die Nachricht *MGetAllPeersForStream* an den Tracker. Diese Nachricht benötigt den Stream Identifier als Parameter und liefert eine Liste mit den Adressen von allen Peers des Streams. Um nun die Struktur erstellen zu können, fragt sie zunächst bei jedem Peer seine Elternknoten

[1] http://jung.sourceforge.net/
[2] http://code.google.com/apis/protocolbuffers/

ab. Dies wird mit Hilfe der Nachricht *GetParents* realisiert welche der Peer mit einer Zuordnung der Elternknoten für jeden Stripe beantwortet. Anschließend wird die Struktur überprüft indem die Kinderknoten von allen Peers sowie der Source abgefragt werden. Dafür wird die Nachricht *GetChildren* verwendet welche für jeden Stripe eine Liste von Kinderknoten zurückliefert. Die Informationen die in beiden Abfragen übereinstimmen werden in den Stream gespeichert und inkonsistente Informationen werden in der Log ausgegeben. Des Weiteren werden die Peers, die nicht auf die Anfragen reagiert haben, aus dem Stream gelöscht und ebenfalls in der Log als tote Peers ausgegeben.

Für die Kommunikation mit den Peers bzw. der Source stellt sie die Methoden *updatePeerInfo(Peer)*, *sendPeerProperties(Peer)*, *killPeer(Peer)*, *terminatePeer(Peer)* bereit. Alle diese Methoden benötigen den Peer als Parameter.

Hierbei aktualisiert die Methode *updatePeerInfo(Peer)* die Eigenschaften des Peers. Dazu sendet sie die Nachricht *GetInfo* an den Peer. Dieser antwortet mit einer Liste der Eigenschaften. Anschließend werden die Eigenschaften in das Peer-Objekt gespeichert. Die Methode *sendPeerProperties* sendet die Eigenschaften, die der Benutzer festgelegt hat, an den Peer. Dazu wird die Nachricht *SetProperties*, welche die Eigenschaften beinhaltet, an den Peer gesendet.

Mit der Methode *killPeer(Peer)* kann einem Peer befohlen werden das Netzwerk sofort zu verlassen. Dazu wird die Nachricht *Kill* an diesen gesendet welche daraufhin sofort das Netzwerk verlässt ohne sich bei den anderen Peers oder dem Tracker abzumelden.Analog dazu existiert die Methode *terminatePeer*. Hierbei wird die Nachricht *Term* an den Peer gesendet welcher daraufhin ebenfalls das Netzwerk verlässt. Allerdings meldet sich hierbei der Peer bei den anderen Peers sowie beim Tracker ab.

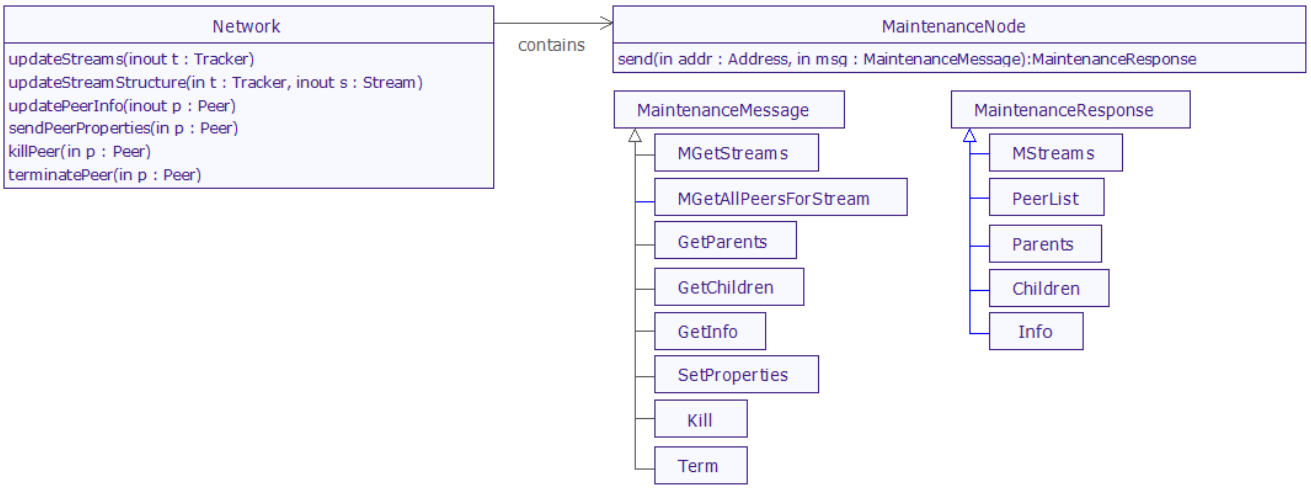

Abbildung 8: Vereinfachtes Klassendiagramm der Netzwerkkommunikation

6.3 Graphen-Framework

Für die Visualisierung des Graphen wird die Klasse *VisualizationViewer* des JUNG-Frameworks verwendet. Diese Klasse arbeitet mit Interfaces um die Darstellung anzupassen sowie vom Anwender ausgelöste Ereignisse zu bearbeiten. Daher wurde für diese Klasse mehrere Implementierungen dieser Interfaces geschrieben. Diese Plugins werden in den folgenden drei Abschnitten vorgestellt. Die Hierarchie dieser Interfaces wird auch in der Abbildung 9 dargestellt.

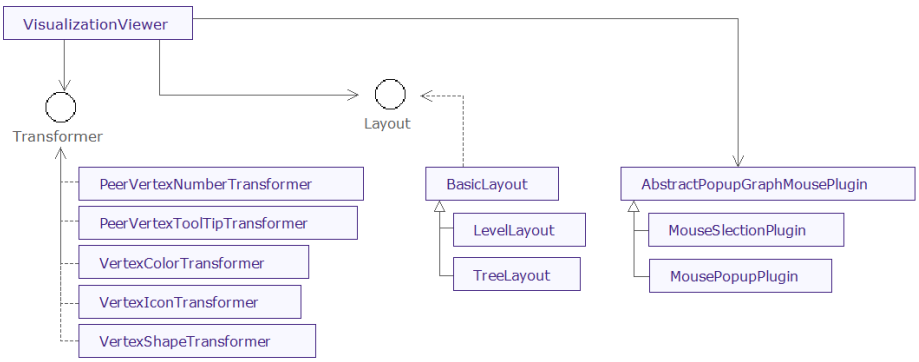

Abbildung 9: Klassenhierarchie für die Graphendarstellung

6.3.1 Implementierung des Maus-Interfaces

Diese Plugins erben von der Klasse *AbstractPopupGraphMousePlugin* und besitzen die Methode *mouseClicked*. Sie werden der *GraphMouse* des VisualizationViewer hinzugefügt. Bei ihnen wird die Methode *mouseClicked* immer dann aufgerufen, wenn auf den Bereich des VisualizationViewer geklickt wurde. Sie befinden sich im Paket *lso.monitor.graph.plugins*.

Um die Peers mit einem Mausklick auszuwählen gibt es die Klasse *MouseSelectionPlugin*. Bei einem Klick wird überprüft ob mit der linken Maustaste auf einen Peer geklickt wurde. Falls dies der Fall ist, wird die Selektion von diesem Peer negiert.

Die Klasse *MousePopupPlugin* überprüft ob mit der rechten Maustaste auf einen Peer geklickt wurde. Sollte dies zutreffen, wird ein Kontextmenü erstellt welches die Optionen bietet den Peer auf verschiedene Weise zu beenden.

6.3.2 Transformatoren

Diese Klassen befinden sich im Paket *lso.monitor.graph.transformers*. Sie implementieren das Interface *Transformer* und werden vom JUNG-Framework als Transformatoren bezeichnet. Ihre Aufgabe ist es eine Zuordnung zwischen einem Peer und einem bestimmten anderen Objekt, wie zum Beispiel einer Zahl im *PeerVertexNumberTransformer*, bereitzustellen. Sie werden entweder direkt dem VisualizationViewer oder seinem *RenderContext* hinzugefügt. Sie befinden sich im Paket *lso.monitor.graph.transformers*.

Hierbei wurden die folgenden Klassen implementiert:

PeerVertexNumberTransformer Sie gibt für den Peer seine eindeutige Nummer innerhalb des Streams zurück.

PeerVertexToolTipTransformer Mit dieser Klasse wird für jeden Peer die Adresse des Peers zurückgegeben.

VertexColorTransformer Da jedem Peer, wenn er ausgewählt ist, eine eindeutige Farbe zugeordnet wird, ist diese Klasse für die Konvertierung eines Peers in eine Farbe zuständig. Falls der Peer nicht ausgewählt ist, wird die Farbe Rot zurückgegeben.

VertexIconTransformer Sie überprüft ob zu einem Peertyp ein Bild zugeordnet ist. Falls dies der Fall ist gibt sie für jeden Peer des Typs das zugeordnete Bild zurück. Wenn der Peer auch markiert ist, wird der Hintergrund des Bildes entsprechend der zugeordneten Farbe angepasst.

VertexShapeTransformer Diese Klasse überprüft ebenfalls ob einem Peertyp ein Bild zugeordnet ist. Falls dies zutrifft erzeugt sie eine Form für das zugeordnete Bild und gibt diese zurück. Diese Form wird als Hitbox für den Peer genutzt um bestimmen zu können ob auf diesen geklickt wurde.

6.3.3 Layout

Die Layouts sind ein wichtiger Bestandteil der Implementierung, da sie für jeden Peer die Position für die Darstellung eines Baumes berechnen. Zwar bietet das Framework bereits einige Layouts an. Diese sind aber nicht für den Monitor geeignet, da sie keine Möglichkeit bieten den Graphen des Streams in mehrere Teilbäume aufzuteilen und diese in aufgeteilten Bereichen anzuzeigen. Für die verwendeten Layouts besteht die Basis aus der abstrakten Klasse *BasicLayout* welche eine Implementierung des Interfaces *Layout* vom JUNG-Framework ist. Jedes Layout benötigt zur Berechnung der Position den Stream welcher mit der Methode *setStream(Stream)* gesetzt werden kann. Falls eine bestimmte Anordnung der Stripes gewünscht ist, kann diese mit der Methode *setStripeLayout(int[])* gesetzt werden. Anschließend müssen die Positionen mit einen Aufruf von *calculate()* berechnet werden. Nachdem die Berechnung abgeschlossen ist, können die Positionen der Peers mit der Methode *transform()* abgerufen werden.

Hierbei ist die Funktionsweise der Methode *calculate()* so, dass sie zunächst überprüft ob ein Layout für die Anordnung der Stripes hinterlegt worden ist. Wenn keines hinterlegt wurde, wird das zuletzt genutzte Layout für die entsprechende Anzahl von Stripes genommen. Anschließend wird der Bereich, in dem die Peers positioniert werden sollen, aufgeteilt und in jeden Teilbereich ein Stripe mit Hilfe der Funktion *calculateStripe()* positioniert. Diese Methode ist abstrakt und muss von den Erben implementiert werden.

Ein Erbe der Klasse BasicLayout ist die Klasse *LevelLayout*. Sie implementiert die Methode *calculateStripe()*. Dabei teilt sie zunächst dem Stripe zugewiesenen Bereich in einen oberen und einen unteren Bereich auf. Diese Aufteilung wird auf Basis der Höhe des Hauptbaums sowie die größte Höhe der abgespaltenen Bäume bestimmt. Anschließend werden zunächst die Peers des oberen Bereichs positioniert.

Bei der Positionierung wird zunächst eine Liste mit den Peers erstellt die auf der aktuellen Ebene platziert werden sollen. Diese Liste wird mit der Source als einzigen Eintrag initialisiert. Die Elemente werden nun gleichmäßig über die vorhandene Breite verteilt und sämtliche Kinder sortiert in eine neue Liste gespeichert. Danach wird die aktuelle Höhe angepasst und die neue Liste nach dem gleichen Muster abgearbeitet bis eine leere Liste entsteht. Die Positionierung des unteren Bereichs funktioniert ähnlich nur dass hier die initiale Liste mit allen Wurzeln der abgespaltenen Teilbäume gefüllt wird.

Eine Alternative zum *LevelLayout* ist die Klasse *TreeLayout* welches sich nur beim Positionieren der Elemente unterscheidet. Hier wird ebenfalls mit einer Liste der aktuellen Peers gearbeitet. Dabei wird jedem Peer der Ebene ein gleichgroßer Teil der vorhandenen Breite zugeteilt und jeder Peer in die Mitte seines Bereichs positioniert. Anschließend wird die Höhe angepasst und für jedes Kind des Peers der Vorgang wiederholt wobei die vorhandene Breite für diese auf die Breite des Elternteils eingeschränkt wird.

6.4 Konfiguration

Sämtliche Klassen der Konfiguration befinden sich in dem Paket *lso.monitor.config*. Hierbei ist die Klasse *Config* für das Laden und Speichern der Konfiguration zuständig. Für das Laden

der Konfiguration stellt sie die Methode *load()* zur Verfügung. Diese überprüft zunächst ob die Datei eine valide XML-Datei ist und lädt diese. Anschließend leitet sie die geladenen Information an die Klassen *Layouts*, *PeerProperties* und *PeerImages* weiter. Des Weiteren stellt sie die Methode *save()* bereit welche die Informationen der vorher genannten Klassen sammelt und im XML-Format in die Konfigurationsdatei speichert.

Die Klasse *Layouts* ist für die Verwaltung der Anordnungen für die Stripes zuständig. Hierbei bietet sie die Möglichkeit mit der Methode *addLayout()* ein neues Layout hinzuzufügen. Mit der Methode *getLayouts()* können alle gespeicherten Layouts für eine bestimmte Anzahl von Stripes abgefragt werden. Des Weiteren bietet sie die Möglichkeit mit *setSessionFavorite()* und *getSessionFavorite()* ein bevorzugtes Layouts pro Stripeanzahl zu setzen und abzurufen. Allerdings werden diese nicht abgespeichert. Außerdem existiert auch die Methode *getSquareLayout()* mit der ein einfaches Layout für eine gegebene Anzahl von Stripes erzeugt werden kann. Diese Methode sorgt dafür das gleichmäßig viele Stripes auf der Breite sowie auf die Höhe aufgeteilt werden.

Des Weiteren bietet die Klasse *Layouts* die Möglichkeit für ein gegebenes Layout ein Vorschaubild zu generieren. Dazu wird zunächst ein leeres Bild erzeugt. Anschließend werden die Positionen der Stripes für das gegebene Layout berechnet und an diesen Positionen auf dem Bild ein Kreis erzeugt.

Die Klasse *PeerImages* verwaltet die Bilder die für die verschiedenen Peertypen definiert sind. Diese werden zunächst mit der Methode *loadPeerImages()* geladen und in der Klasse gespeichert. Anschließend können diese mit der Methode *getPeerImage()* abgefragt werden.

Der letzte Bereich der Konfiguration ist für die Verwaltung der bekannten Eigenschaften der Peers zuständig. Diese werden in der Klasse *PeerProperties* gespeichert welche auch die verfügbaren Eigenschaften bereitstellt. Die Eigenschaften sind Objekte der Klasse *PeerProperty*. Diese Klasse speichert den Eigenschaftsnamen, den Typ sowie die zulässigen Werte. Außerdem bietet sie die Methode *getComponentType()* an. Mit dieser kann abgefragt werden welche Eingabemöglichkeit in der Benutzeroberfläche angezeigt werden soll. Falls in der Konfiguration ein Wert definiert wurde, wird dieser zurück gegeben ansonsten wird anhand des Typs der Eigenschaft sowie der zulässigen Werte eine geeignete Eingabemöglichkeit ausgesucht und zurückgegeben.

6.5 Benutzerinterface

Die Klassen des Benutzerinterfaces befinden sich im Paket *lso.monitor.swingui*. Hierbei ist die Klasse *MonitoringServer* die Hauptkomponente des Benutzerinterfaces. Mit ihrer Hilfe wird die Benutzeroberfläche erstellt und angezeigt. Außerdem behandelt sie viele der vom Benutzer ausgeführten Aktionen. Wenn ein Benutzer zum Beispiel ein anderes Layout auswählt wird die Methode *actionPerformed()* aufgerufen welche in diesem Fall das neue Layout an das Objekt der Klasse *StreamVisualization* weitergibt.

Hierbei ist die Klasse *StreamVisualization* eine Schnittstelle zwischen dem JUNG-Framework und der Benutzeroberfläche. Sie benutzt den *VisualizationViewer* des Frameworks sowie die selbst erstellen Plugins für diesen um den Graphen anzuzeigen. Außerdem stellt sie die Funktionalität bereit den Stream automatisch in einem vom Benutzer festgelegtem Intervall zu aktualisieren.

Die Ausgabe der Meldungen über den Netzwerkverkehr sowie von Fehlern und Informationen wird mit der Klasse *LogBox* realisiert. Sie bietet die Methode *addLine()* an um Meldungen für

die Ausgabe hinzuzufügen. Sämtliche Meldungen werden mit einem Zeitstempel versehen und in einem Textfeld ausgegeben.

Die Klassen *LayoutBoxEditor* und *LayoutBoxRenderer* ermöglichen die Vorschaubilder bei der Layoutauswahl. Hierbei ist der *LayoutBoxRenderer* für die Darstellung der Einträge zuständig. Der *LayoutBoxEditor* hingegen überwacht die Eingabe im Feld und erstellt während der Eingabe passende Vorschaubilder und zeigt diese an.

Des Weiteren gibt es für den Bereich, in dem die ausgewählten Peers mit ihren Eigenschaften dargestellt werden, die Klassen *PeerPropertiesPanelManager*, *PeerPropertiesPanel* und *PeerPropertyComponent*. Hierbei ist der *PeerPropertiesPanelManager* für die Verwaltung der ausgewählten Peers zuständig. Er erzeugt für jeden ausgewählten Peer ein neues Objekt vom Typ *PeerPropertiesPanel* welches er seinen Bereich hinzufügt. Wenn ein Peer deselektiert wird löscht er das zugehörige Panel. Außerdem ist er für die Schaltflächen die alle ausgewählten Peers beeinflussen sowie die Schaltfläche um neue Peers hinzuzufügen zuständig.

Das *PeerPropertiesPanel* ist immer für einen Peer zuständig. Es zeigt dabei die Schaltfläche mit dem Bild und Farbe des Peers an sowie sein Name. Des Weiteren beinhaltet es für jede Eigenschaft des Peers ein *PeerPropertyComponent*. Außerdem beinhaltet es die Schaltflächen um die geänderten Eigenschaften zu speichern, die Änderungen zurück zusetzen, den Peer zu Terminieren und den Peer zu töten.

Die Klasse *PeerPropertyComponent* ist für die Darstellung einer Eigenschaft zuständig. Sie beinhaltet ein Feld mit der Beschriftung der Eigenschaft sowie eine Eingabemöglichkeit welche von der Art der Eigenschaft abhängig ist.

7 Nutzung

In diesem Kapitel zeigen wir die Benutzung des Monitors. Dabei wird zunächst mit der Konfiguration des Monitors begonnen. Anschließend stellen wir die Funktionen der Benutzeroberfläche und zum Abschluss die unterstützen Kommandozeilenargumente vor.

7.1 Konfiguration

Die Konfiguration besteht aus einer Baumstruktur, welche im XML-Format gespeichert und geladen wird. Sie beginnt mit einer Defintion der XML-Version sowie der Enkodierung. Anschliessend wird der Wurzelknoten der Konfiguraiton mit dem Tag *MONITOR* definiert. Für diesen Knoten sind zur Zeit die Kinder *LAYOUTS*, *PEER_PROPERTIES*, *PEER_IMAGES* sowie das Attribut *LOG_POSITION* definiert. Hierbei gibt das Attribut *LOG_POSITION* die Position der Netzwerkausgabe in der Benutzeroberfläche an. Es kann den Wert *TOP* für eine Anzeige am oberen Rand des Fensters sowie den Wert *RIGHT* für die Anzeige am rechten Rand des Fensters annehmen. Die grundlegende Struktur sieht somit folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<MONITOR>
  <NODE_IMAGES>
...
  </NODE_IMAGES>
...
  </LAYOUTS>
  <PEER_PROPERTIES>
...
  </PEER_PROPERTIES>
  <LOG_POSITION>TOP</LOG_POSITION>
</MONITOR>
```

Die Sektion *LAYOUTS* ist für die Definiton der Layouts für die Anordnung der Stripes im Graphen zuständig. Jedes Kind dieses Elements ist eine Sammlung von Layouts für eine bestimmte Anzahl von Stripes. Die Namen der Kinder sind definiert als *LAYOUT* konkateniert mit der Anzahl der Stripes, zum Beispiel *LAYOUT3*. In diesen Knoten befinden sich die einzelnen Layouts, welche den Tag *LAYOUT* besitzen. Diese haben als Inhalt eine Komma-getrennte Liste von Zahlen. Dabei steht jede Zahl für die Anzahl von Stripes die in einer Reihe dargestellt werden. In dem folgenden Beispiel werden Layouts für Streams mit 3 sowie mit 6 Stripes definiert. Hierbei wird für Streams mit 3 Stripes die Layouts *1,2*, *1,1,1* und *2,1* festgelegt sowie für Streams mit 6 Stripes das Layout *1,2,3* definiert.

```
<LAYOUTS>
  <LAYOUT3>
    <LAYOUT>1,2</LAYOUT>
    <LAYOUT>1,1,1</LAYOUT>
    <LAYOUT>2,1</LAYOUT>
  </LAYOUT3>
  <LAYOUT6>
    <LAYOUT>1,2,3</LAYOUT>
  </LAYOUT6>
</LAYOUTS>
```

Die Sektion *PEER_IMAGES* definiert die Bilder für einen Peertyp. Die Kinder dieses Elements haben als Tag den Nodetypen. Der Inhalt jedes Kindes ist die Angabe eines relativen Pfades zu einem Bild. Falls für einen Peertypen kein Bild definiert ist oder es nicht gefunden wurde, werden diese Elemente mit Hilfe eines Kreises dargestellt. Die folgenden Angaben definieren die Bilder für Peers die vom Typ *LINUX* oder vom Typ *WINDOWS* sind.

```
<PEER_IMAGES>
  <LINUX>img/linuxalpha.png</LINUX>
  <WINDOWS>img/windows-logoalpha.png</WINDOWS>
</PEER_IMAGES>
```

Der Bereich *PEER_PROPERTIES* ist für die Definiton von bekannten Eigenschaften der Peers zuständig. Hierbei wird für jede Definiton ein Element mit dem Namen *PROPERTY* erstellt. Diese Elemente unterstützen die folgenden Attribute:

NAME Dies definiert den Namen der Eigenschaft welche vom Peer übertragen wird und ist ein Pflichtfeld für jedes Property.

TYPE Hiermit wird definiert welche Werte die Eigenschaft speichern kann und ist genauso wie *NAME* immer benötigt. Zur Zeit werden Zeichenketten (*STRING*), Zahlen(*NUMBER*) und Wahrheitswerte(*BOOLEAN*) unterstützt.

DISPLAY_NAME Dies ist eine optionale Eigenschaft und definiert den Anzeigenamen welcher vom Benutzerinterface verwendet wird.

MIN, *MAX*, *STEPS* Diese drei Eigenschaften werden unterstützt wenn das Property eine Zahl ist. Sie sind optional und legen das Minimum, Maximum sowie die Schrittgröße für das Property fest.

COMPONENT Wenn man selber bestimmen möchte welches Komponente zur Darstellung bzw. zur Bearbeitung einer Eigenschaft genutzt wird, kann man diese Eigenschaft benutzen. Dabei sind die folgenden Werte gültig: TEXTFIELD, NUMBERFIELD, COMBOBOX, BOOLEAN, HIDDEN.

DEFAULT Hiermit kann ein Standardwert für eine Eigenschaft definiert werden, welche genutzt wird, wenn diese Eigenschaft für einen Peer nicht festgelegt ist.

READONLY Eigenschaften mit diesem Attribut können in der Benutzeroberfläche nicht geändert werden.

In dem folgenden Beispiel werden vier Eigenschaften definiert, welche die Namen *NAME*, *MOBILE*, *MAINTENANCE_ADDRESS* und *BANDWIDTH* haben. Dabei ist festgelegt, dass die ersten drei Eigenschaften nicht geändert werden können. Die Eigenschaft *BANDWIDTH* ist als Zahl festgelegt mit einem minimalen Wert von 200 und einem maximalen Wert 1000 sowie einer Schrittweite von 200. Des Weiteren wurde festgelegt das diese Eigenschaft mit einem Drop-Down-Feld angezeigt wird.

```
<PEER_PROPERTIES>
  <PROPERTY>
    <NAME>NAME</NAME>
    <TYPE>String</TYPE>
    <DISPLAY_NAME>Name</DISPLAY_NAME>
    <READONLY>true</READONLY>
  </PROPERTY>
  <PROPERTY>
    <NAME>MOBILE</NAME>
    <TYPE>Boolean</TYPE>
    <DISPLAY_NAME>Mobil?</DISPLAY_NAME>
    <READONLY>true</READONLY>
  </PROPERTY>
  <PROPERTY>
```

```
    <NAME>MAINTENANCE_ADDRESS</NAME>
    <TYPE>String</TYPE>
    <DISPLAY_NAME>Adresse</DISPLAY_NAME>
    <READONLY>true</READONLY>
  </PROPERTY>
  <PROPERTY>
    <NAME>BANDWIDTH</NAME>
    <TYPE>Number</TYPE>
    <DISPLAY_NAME>Bandbreite</DISPLAY_NAME>
    <STEPS>200.0</STEPS>
    <MIN>200.0</MIN>
    <MAX>1000.0</MAX>
    <COMPONENT>COMBOBOX</COMPONENT>
  </PROPERTY>
</PEER_PROPERTIES>
```

7.2 Benutzeroberfläche

Die Benutzeroberfläche lässt sich in die vier Teile Einstellungen, Eigenschaften selektierter Komponenten, Logausgabe und Visualisierung aufteilen. Diese wird in Abbildung 10 dargestellt.

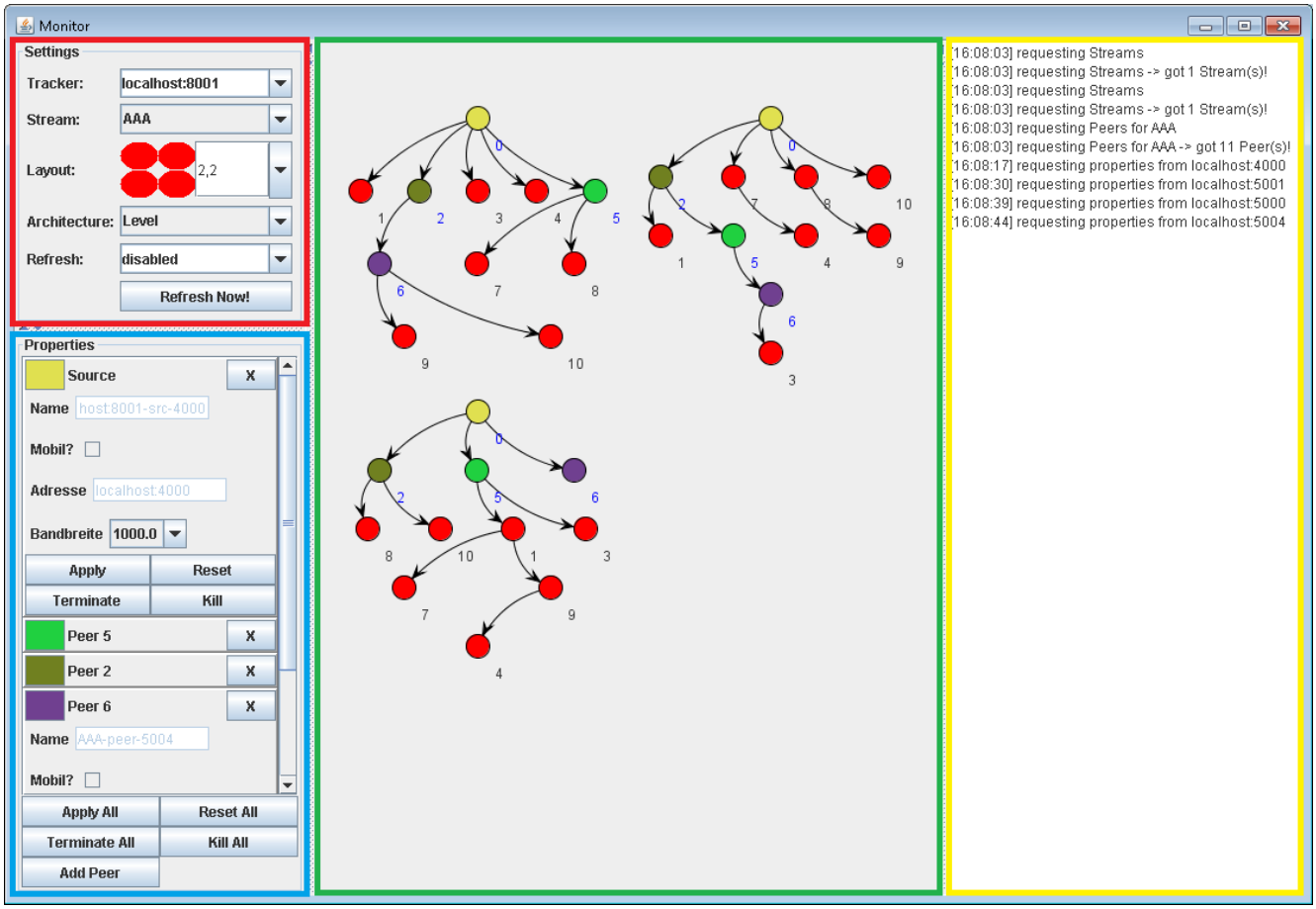

Abbildung 10: Benutzeroberfläche bestehend aus den Einstellungen, Eigenschaften selektierter Komponenten, Visualisierung und Logausgabe

7.2.1 Einstellungen

Im Bereich Einstellungen kann der momentan genutzte Tracker sowie der anzuzeigende Stream eingestellt werden. Außerdem befinden sich hier auch die Möglichkeiten das Layout der

angezeigten Graphen zu ändern sowie die automatische Aktualisierung einzustellen. Dies ist in Abbildung 10 der obere linke Bereich.

Die erste Option ist die Eingabe des Trackers. Die Eingabe des Trackers muss mit dem Hostname oder der IP-Adresse beginnen und mit einem Doppelpunkt sowie der Port-Nummer enden. Nach der Eingabe muss diese mit der Entertaste bestätigt werden. Danach wird der Tracker kontaktiert und die Stream-Liste, welche die zweite Option ist, aktualisiert. In dem Drop-Down-Menü Stream befinden sich alle vom Tracker verfügbaren Streams. Sobald ein Stream ausgewählt wird, wird die Peer-Struktur des Streams aktualisiert und dargestellt.

Die nächste Option *Layout* beeinflusst die Anordnung der Stripes im Graphen. Hierbei ist zunächst eine Vorschau des aktuellen Layouts zu sehen. Nach einem Klick auf das Drop-Down-Menü bekommt man eine Übersicht der gespeicherten Layouts für die Anzahl von Stripes die der aktuelle Stream hat. Nach Auswahl eines Layouts wird der Graph an das neue Layout angepasst. Außerdem besteht die Möglichkeit in diesem Feld ein neues Layout einzugeben. Hierbei wird die Anzahl der Stripes für die jeweiligen Reihen mit Komma getrennten Zahlen angegeben. Wenn zum Beispiel in der ersten Reihe ein Stripe, in der zweiten Reihe zwei Stripes und in der dritten Reihe ein Stripe stehen soll, muss die Eingabe *1,2,1* lauten. Während der Eingabe wird die Vorschau für das eingegebene Layout aktualisiert. Die Eingabe muss mit der Entertaste bestätigt werden. Sobald die Eingabe bestätigt ist, wird das Layout gespeichert und der Graph an das neue Layout angepasst.

Die darauf folgende Option nennt sich *Architecture*. Diese beeinflusst die Anordnung der Peers innerhalb eines Baumes. Nach einem Klick auf das Drop-Down-Feld gibt es die Auswahl zwischen den Optionen *Level* und *Tree*. Bei der Option *Level* werden die Peers, welche auf der gleichen Höhe im Baum sind, auf die gesamte Breite aufgeteilt. Bei der Auswahl *Tree* werden die Peers nur auf die Breite ihrer Eltern aufgeteilt. Nach der Auswahl einer Option wird der Graph angepasst und dargestellt. Abbildungen 11 und 12 zeigen das gleiche Netzwerk mit den verschiedenen Architekturen.

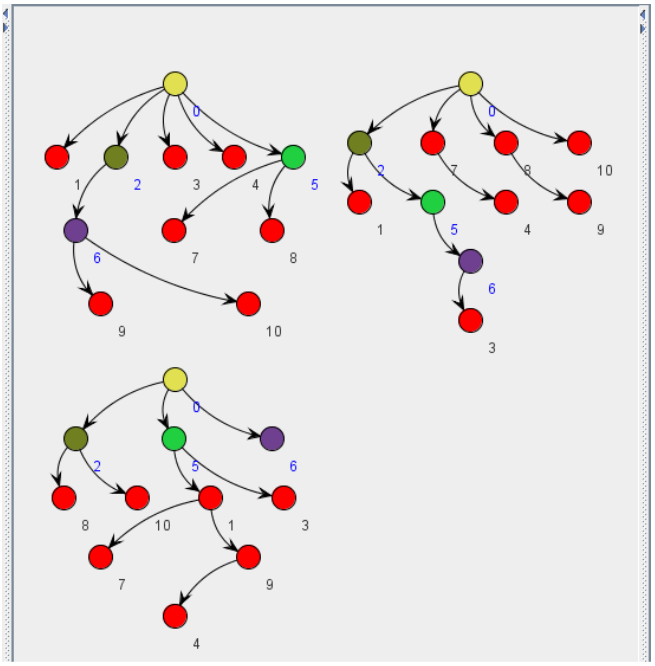

Abbildung 11: Overlay Topologie mit Level-Architektur

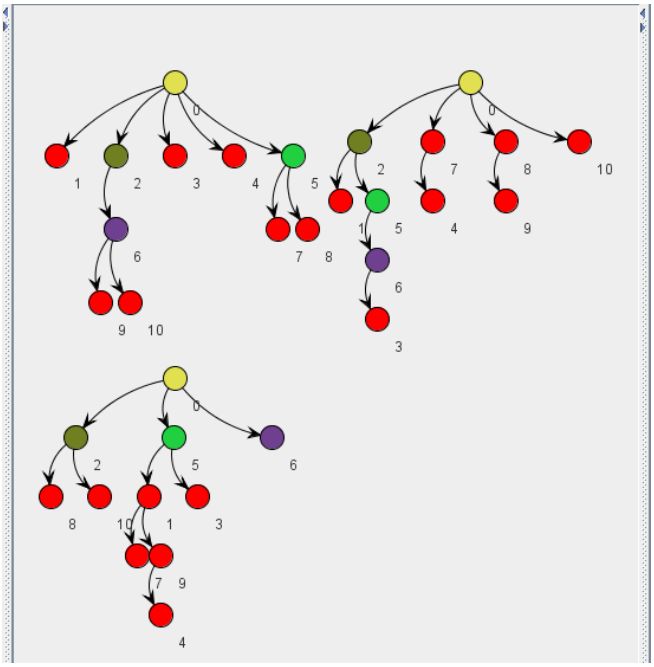

Abbildung 12: Overlay Topologie mit Tree-Architektur

Die letzte Option im Bereich Einstellungen ist das Feld *Refresh*. Mit dieser Einstellung kann das automatische Aktualisieren des Graphen angepasst werden. Hier gibt es ebenfalls die Möglichkeit eine Zeit aus der Drop-Down-Liste zu wählen oder sie manuell einzugeben. Falls die Zeit manuell eingegeben wird, muss die Zahl mit der Endung *ms* für Millisekunden oder *s* für Sekunden enden und mit der Entertaste bestätigt werden. Falls keine korrekte Eingabe erfolgt ist oder *disabled* aus der Drop-Down-Liste gewählt wurde, wird die automatische Aktualisierung ausgeschaltet. Wenn die automatische Aktualisierung eingeschaltet ist, wird die Peer-Struktur der Eingabe entsprechend aktualisiert. Mit der Schaltfläche *Refresh Now!* unterhalb der Auswahlliste kann eine Aktualisierung sofort ausgeführt werden. Allerdings ist diese Schaltfläche nur aktiv wenn die automatische Aktualisierung ausgeschaltet ist.

7.2.2 Eigenschaften selektierter Komponenten

In diesem Bereich befinden sich die ausgewählten Peers. Dabei werden die Eigenschaften der ausgewählten Peers angezeigt und es besteht die Möglichkeit diese anzupassen oder auch die Peers zu terminieren.

In Abbildung 10 ist dieser Bereich in der unteren linken Ecke dargestellt. Hier sieht man, dass vier Peers *Source*, *Peer 5*, *Peer 2* und *Peer 6* ausgewählt sind. Links neben den Namen des Peers wird ein Feld mit der Farbe angezeigt mit welcher der Peer im Graphen markiert ist. Wenn man auf dieses Feld klickt wird eine neue Farbe für den Peer per Zufall generiert. Rechts neben dem Namen des Peers befindet sich ein Button zur Deselektierung des Peers. Wenn auf den Namen des Peers geklickt wird, werden seine Eigenschaften und Schaltflächen sichtbar beziehungsweise unsichtbar gemacht. Unter dem Namen des Peers beginnt die Aufzählung mit den Eigenschaften des Peers. Hierbei steht auf der linken Seite immer die Bezeichnung der Eigenschaft und auf der rechten Seite das Feld mit dem aktuellen Wert. Falls dieses Feld nicht ausgegraut ist kann der Wert geändert werden. Sobald ein Wert in einem Feld geändert wurde, werden die Schaltflächen *Apply* und *Reset* sowie die globalen *Apply All* und Reset All aktiviert. Bei einem Klick auf *Apply* werden die geänderten Eigenschaften an den Peer gesendet.Analog dazu setzt die Schaltfläche *Reset* die Änderungen zurück sofern sie noch nicht übertragen worden sind. Mit der Schaltfläche *Terminate* kann einem Peer befohlen werden das Netzwerk ordnungsgemäß zu verlassen. Im Gegensatz dazu befiehlt die Schaltfläche *Kill* dem Peer das Netzwerk spontan zu verlassen. Am unteren Ende des Bereichs befinden sich noch die Schaltflächen *Apply All, Reset All, Terminate All, Kill All*. Diese führen die jeweilige Aktion für alle ausgewählten Peers aus.

7.2.3 Logausgabe

Die Ausgabe von Log-Meldungen ist in Abbildung 10 im gelbem Bereich zu sehen. Diese Meldungen basieren auf Ereignissen wie zum Beispiel das Senden und Empfangen von Nachrichten. Des Weiteren werden auch Inkonsistenzen des Netzwerkoverlays, welche bei der Abfrage von Eltern und Kindern der Peers sowie der Source entdeckt werden, hier ausgegeben.

7.2.4 Visualisierung

In diesem Bereich wird die Struktur des aktuellen Streams visualiert. Er wird in Abbildung 11 genauer dargestellt.

Der Bereich wird entsprechend dem Stripe Layout aufgeteilt und die Stripes in ihrem jeweiligen Bereich gezeichnet. Hierbei wird für jeden Stripe zunächst der Teilbaum welcher die Quelle als Wurzel hat gezeichnet. Falls es Peers gibt, die nicht mit dem Teilbaum der

Quelle verbunden sind, werden diese unterhalb des Teilbaums der Quelle gezeichnet. Wenn diese Peers untereinander Verbindungen haben, werden sie ebenfalls als Bäume dargestellt. Jeder Peer bekommt eine für den Stream einzigartige ID zugewiesen um die Erkennung der Peers in den verschiedenen Stripes zu ermöglichen. Bei einem Linksklick auf einen Knoten wird der zugehörige Peer markiert. Die Markierung wird optisch mit einer zufällig generierten Farbe in allen Stripes hervorgehoben. Des Weiteren wird der Peer im Auswahlbereich aufgelistet. Bei einem Rechtsklick auf einen Peer erscheint ein Menü mit den Optionen den Peer zu terminieren oder ihn zum Absturz zu zwingen. Falls eine der beiden Optionen ausgewählt wird, wird die Nachricht an den Peer gesendet und die Struktur des Netzwerks aktualisiert.

In Abbildung 13 wird das gleiche Netzwerk dargestellt wie in Abbildung 12 nach dem spontanen Verlassen des Peers 2. Diese Graph besteht aus drei Stripes und die Peers 5 und 6 sowie die Source sind ausgewählt. Da der Peer 2 Kinder hatte, sind diese nun von den Daten der Stripes, bei denen sie ein Kind von ihm waren, abgetrennt. In den Stripes oben führte das Verlassen zu einer Abspaltung von Teilbäumen. In diesem Fall sind die Peers 6 und 10 sogar von zwei Stripes agebschnitten. Die Abbildung 14 zeigt einen Graphen mit Peers an bei denen Bilder für diese hinterlegt waren. Bei diesem sind die Peer 7, 9 und 10 ausgewählt.

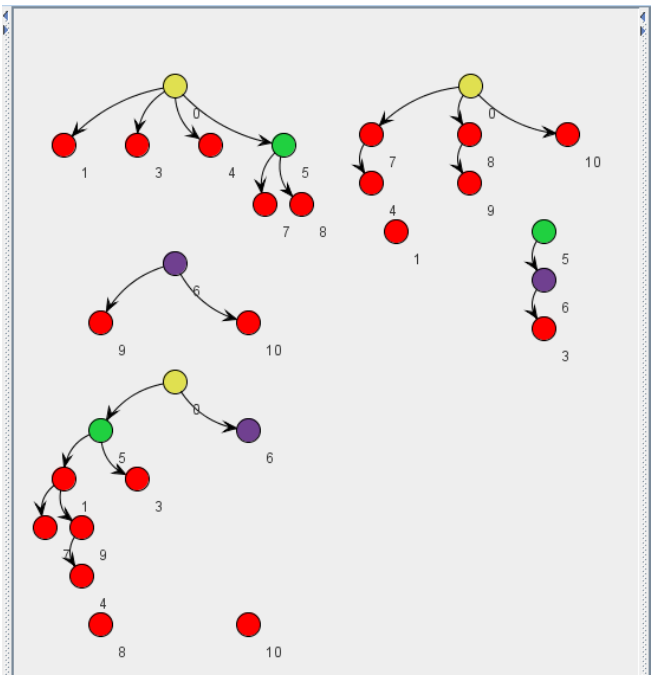

Abbildung 13: Graph nach spontanem Verlassen des Peers 2

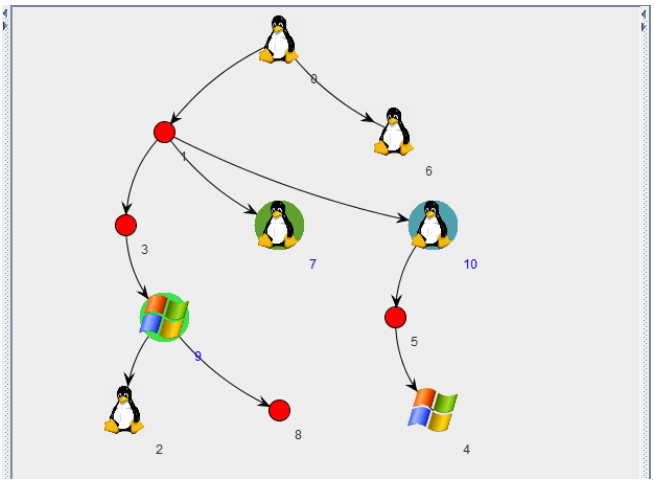

Abbildung 14: Darstellung eines Graphen mit für Peers hinterlegeten Bildern

7.3 Kommandozeilenparameter

Der Monitor bietet die Möglichkeit einige Einstellungen mit Hilfe eines Kommandozeilenbefehls einzustellen. Die einzelnen Parameter werden mit einem doppelten Minus zusammen mit dem Namen des Parameters angegeben. Anschließend folgt ein Leerzeichen und der Wert des Parameters. Hierbei werden die folgenden optionalen Kommandozeilenparameter unterstützt:

bind Mit dieser Option kann festgelegt werden welche Adresse der Monitor zum Empfangen von Nachrichten benutzt.

tracker Hiermit kann festgelegt werden welcher Tracker initial benutzt wird.

Beispiel für einen Kommandozeilenbefehl welcher den Monitor den lokalen Port 5001 zuweist und als initialen Tracker den lokalen Tracker auf Port 8001 angibt:

```
java -jar monitor.jar -bind localhost:5001 -tracker localhost:8001
```

8 Zusammenfassung und Ausblick

Durch die steigende Beliebtheit von Live-Streaming und der dadurch gestiegene Anzahl von Nutzern, werden P2P-basierte Live-Streaming-Systeme wegen ihrer guten Skalierbarkeit immer intressanter. So wurde auch an der TU Darmstadt ein solches System implementiert. Da diese Systeme ohne Hilfsmittel nur schwer zu analysieren wurde mit dieser Arbeit ein Monitor für dieses System implementiert. Dieser hilft beim Analysieren sowie Testen des Systems.

Bei der Entwicklung und Implementierung wurden die Anforderungen an dieses System erfüllt. Die Anforderungen bestanden dabei zunächst aus einer Visualisierung des Netzwerks sowie der Manipulation von Peers. Dabei sollte es auch möglich seien das Benutzerinterface zu konfigurieren.

Durch den Monitor kann das Netzwerk gut visualisiert werden womit man schnell einen Überblick über das gesamte Netzwerk und seine Struktur hat. Dabei werden auch Fehler in der Struktur aufgezeigt. Dies kann sehr nützlich sein um Fehler in Algorithmen zu entdecken und die Adaption der Algorithmen an neue Situationen im Netzwerk beurteilen zu können.

Des Weiteren besteht nun die Möglichkeit direkt in das Netzwerk einzugreifen und somit bestimmte Peers auf verschiedene Arten zu beenden und somit verschiedene Situationen im Netzwerk zu testen. Es besteht jetzt ebenfalls die Möglichkeit Eigenschaften der Peers anzupassen und somit zum Beispiel das Verhalten eines Peers zu testen, wenn seine Bandbreite eingeschränkt wird und er somit nicht mehr genügend Bandbreite für seine Kinder zur Verfügung hat.

Es werden bereits über einige Erweiterungen für den Monitor nachgedacht. Zu diesen gehören zum Beispiel die folgenden:

Timeline Mit Hilfe einer Timeline könnte man die Veränderung des Overlays genauer analysieren und somit auch schnelle Änderungen im Overlay besser beobachtet werden. Zur Integration dieses Features müsste eine neue Klasse entwickelt werden welche die Änderungen des Overlays speichert und auch Änderungen am Overlay für die Darstellung wieder rückgängig machen kann. Des Weiteren muss in die Oberfläche weitere Element(e) zur Kontrolle der Timeline-Funktion geschaffen werden. Dafür könnte z.B. ein Slider in die StreamVisualization-Klasse integriert werden.

Android Ein weiteres intressantes Feature wäre die Integration einer GUI-Oberfläche für das Android System. Allerdings wird das Swing-Framework nicht von Android unterstützt und es müsste daher ein anderes Framework verwendet werden.

Literatur

[ADS+11] Osama Abboud, Niloofar Dezfuli, Benjamin Schiller, Dirk Schnelle-Walka, and Thorsten Strufe. Interim Report of Phase II - Development of Social TV Service Technologies based on Next Generation IPTV Infrastructures. Interim Report, TU Darmstadt, 2011.

[HLR08] X. Hei, Y. Liu, and K.W. Ross. Iptv over p2p streaming networks: the mesh-pull approach. *Communications Magazine, IEEE*, 46(2):86–92, 2008.

[JGJ+00] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: reliable multicasting with on overlay network. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 14–14, Berkeley, CA, USA, 2000. USENIX Association.

[Str07] T. Strufe. *A Peer-to-Peer-based Approach for the Transmission of Live Multimedia Streams (German:"Ein Peer-to-Peer-basierter Ansatz für die Live-Übertragung multimedialer Daten")*. PhD thesis, TU Ilmenau, 2007.

[XLKZ07] S. Xie, B. Li, G.Y. Keung, and X. Zhang. Coolstreaming: Design, theory, and practice. *Multimedia, IEEE Transactions on*, 9(8):1661–1671, 2007.