Farsight

Discovery of Distant SeMoA Hosts

Jan Hävecker, April 2004

Inhaltsverzeichnis

1 Einleitung

In der vorliegenden Arbeit soll ein Modul für das SeMoA-System vorgestellt werden. SeMoA steht für *Secure Mobile Agents*. Es handelt sich dabei um eine erweiterbare und offene Ablaufumgebung für mobile Agenten [1]. Innerhalb eines Agentensystems führen Agenten geforderte Aufträge aus. Agenten lassen sich als Systeme beschreiben, die in der Lage sind autonom, sinnvolle Handlungen in einer bestimmten Umgebung auszuführen. Mobile Agenten sind weiterhin in der Lage das System in dem sie arbeiten zu verlassen und in einem anderen geeigneten Agentensystem ihre Arbeit fortzuführen. Man spricht dabei von *Migration*.

Aufgabe des Moduls soll es sein, Kontakte zu anderen SeMoA-Servern zu verwalteten und diese unter Umständen auch automatisch aufzuspüren. Das Entdecken von SeMoA-Servern in verschiedenen Netzwerkumgebungen wird im Rahmen dieses Textes als *Host-Discovery* bezeichnet.

Eine effektive Methode zur Host-Discovery ist von größter Bedeutung für ein System, das mit mobilen Agenten arbeitet. Damit ein Agent seine Mobilität überhaupt nutzen kann, muss dieser ein oder mehrere, mögliche Ziele kennen. Im Fall von SeMoA sind diese Ziele Computersysteme, auf denen ein SeMoA-Server ausgeführt wird. Die Kenntnis über die Existenz, Adresse und momentane Verfügbarkeit dieser Systeme ist allgemein nicht gegeben.

SeMoA verfügt über ein Host-Discovery-Modul, welches nur innerhalb der Grenzen eines LAN arbeitet. Es trägt den Namen *Vicinity*. Vicinity ist in der Lage, SeMoA-Server innerhalb der Grenzen eines LAN zu entdecken, was jedoch für verteilte Anwendungen meist nicht ausreicht. Das neue Modul soll über die Grenzen des LAN hinweg arbeiten können, weshalb es auf den Namen *Farsight* getauft wurde.

Diese Ausarbeitung ist in drei Teile gegliedert. Im ersten Teil *Host-Discovery* wird der Begriff Host-Discovery genauer betrachtet. Dabei werden einige verwandte Anwendungen und Arbeiten zum diesem Thema vorgestellt. Im zweiten Teil *Analyse und Lösungsansatz* wird die

Idee zur Lösung des Problems der Host-Discovery erläutert. Der dritte Teil, bestehend aus den Kapiteln *Implementierung*, sowie *Installation und Betrieb*, beschreibt die Funktionsweise und Architektur des vorliegenden Farsight-Moduls.

2 Host-Discovery

Im Rahmen dieser Arbeit bezeichnet Host-Discovery den Vorgang etwas über die Existenz von SeMoA-Servern in anderen, erreichbaren Teilen eines Netzwerkes zu erfahren.

In großen, verteilten Computer-Netzwerken kommt es oft vor, dass ein Teil der Rechner kooperieren möchte um gemeinsam eine bestimmte Aufgabe zu bearbeiten. Dafür müssen diese Rechner aber zunächst einmal wechselseitig von ihrer Existenz wissen. Anders ausgedrückt: Sie müssen die Frage „Wer im Netz will mit mir zusammenarbeiten?“ beantworten können. Dieser erste Schritt wird im in einem allgemeineren Kontext als *Resource Discovery* bezeichnet [7].

Eine große Hürde ist dabei der *initiale Kontakt*. Ist erstmal ein Teilnehmer eines bestimmten Netzwerks gefunden, so kennt dieser in der Regel andere Teilnehmer, deren Adressen er weitervermitteln kann. Danach ist das Netz theoretisch in der Lage sich selbst zu organisieren. Zu lösen ist das Problem woher man jene ersten Kontakte kennt. Diese Problematik ist eine Ausprägung des klassischen *Henne-Ei-Problems*. Wen gab es zuerst, die Henne oder das Ei? Ohne Teilnehmer kann es kein Netz geben und ohne ein bekanntes Netz können keine neuen Teilnehmer ins Netz integriert werden.

Im Rahmen der Planungsarbeiten für die Implementierung von Farsight wurden einige verwandte Gebiete bezüglich deren Methoden zur Resource-Discovery untersucht, die im Folgenden dargestellt werden.

2.1 LAN

Lokale Netzwerke, die Ethernet oder Token-Ring Technologien einsetzen, bieten a priori zwei charakteristische Dienste: Vollständige Konnektivität und Broadcasting.

Vollständige Konnektivität bedeutet, dass jeder Rechner Pakete an jeden anderen senden kann.

Broadcasting meint, dass ein Rechner ein einzelnes Paket verschicken kann, das alle anderen Rechner im LAN erreicht (*broadcast*) oder alle Rechner in einer bestimmten Gruppe (*multicast*).

Das Thema LAN soll an dieser Stelle nicht weiter vertieft werden. In [8] finden sich weiterführende Informationen zum den Themen Broadcasting und Multicasting, sowie LAN im Allgemeinen.

2.2 Vicinity

Vicinity ist ein SeMoA-Modul, das innerhalb der Grenzen eines LAN Host-Discovery betreibt. Es bedient sich dabei der Multicasting-Technologie.

Die Gruppe, in der kommuniziert werden soll, wird durch eine Menge von IP-Adressen mit zugehörigem Port eindeutig festgelegt. Unter allen Vicinity-Diensten innerhalb der gleichen Gruppe wird automatisch ein Koordinator gewählt. Dieser ist dafür verantwortlich neue Kontakte zu finden, und deren Existenz an die anderen Mitglieder der Gruppe weiterzuleiten. Vom Koordinator werden in regelmäßigen Abständen Ping-Pakete an die Gruppenadresse gesendet. Alle Gruppenmitglieder lauschen auf dieser Adresse und senden jeweils ein Pong-Paket zurück. Damit signalisieren sie, dass sie der Gruppe angehören und noch erreichbar sind. Auf diese Weise werden auch neu hinzugekommene Mitglieder entdeckt und mittels eines Update-Paketes publiziert. Sollte ein bereits bekannter Kontakt dem Koordinator nicht mehr auf ein Ping-Paket antworten, so wird er aus der Kontakt-Liste gelöscht. Dies wird auch den anderen Gruppenmitgliedern mittels eines Update-Paketes mitgeteilt. Sollte der Koordinator selbst nicht mehr verfügbar sein, so wird automatisch ein neuer Koordinator unter den verbleibenden Gruppenmitgliedern gewählt. Das System ist robust und arbeitet effizient. Da aber Broadcasting-Pakete in der Regel nicht aus dem LAN hinaus geleitet werden, bleibt der Einsatz von Vicinity auf das LAN beschränkt.

2.3 Internet

Im Internet werden von den Routern verschiedene Algorithmen zur Resource-Discovery eingesetzt. Das BGP (*Border Gateway Protocol*) [11], welches für *interdomain routing* eingesetzt wird, verwendet unter Anderem das *Interior Gateway Protocol* OSPF (*Open Shortest Path First*) [8,12], um die

Erreichbarkeit von anderen autonomen Systemen (Netzwerken) zu überprüfen. Das Routing-Protokoll OSPF verwendet seinerseits den *Flooding-Algorithmus* (*FA*) [7,8] zur Resource-Discovery.

Im FA wird vorausgesetzt, dass ein Rechner so konfiguriert ist, dass er eine feste Anzahl von benachbarten Rechnern bereits kennt. Nur mit diesen Rechnern darf direkt kommuniziert werden. Modelliert man das Netz als einen Graph, so darf ein Knoten also nur über Kanten kommunizieren, die bereits von Anfang an vorhanden waren; neu hinzugefügte Kanten werden nicht benutzt. Dabei sind die Kanten, die zu den initialen Nachbarn führen nicht notwendigerweise equivalent mit den dem Netzwerk zu Grunde liegenden physikalischen Verbindungen. Sie sind eher als virtuelle Verbindungen zu betrachten, die jeweils einen Pfad von Verbindungen der physikalischen Ebene repräsentieren. Weiterhin muss der Graph, mit dem der FA beginnt, wenigstens schwach zusammenhängend sein (d.h. zusammenhängend wenn man alle gerichteten Kanten durch ungerichtete ersetzt).

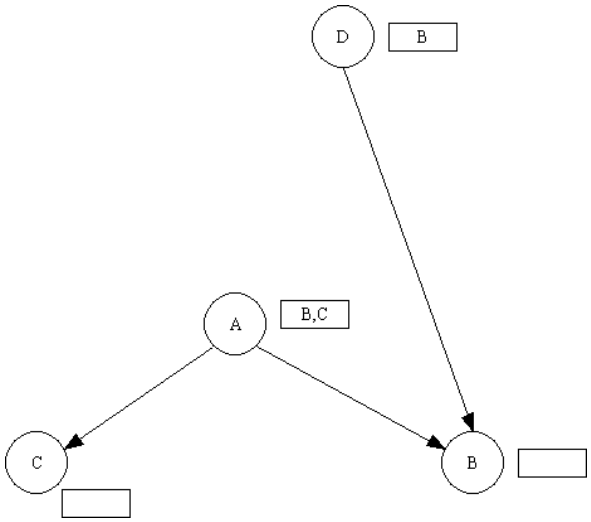


Abbildung 1

Ausgangsituation: Ein schwach zusammenhängender, gerichteter Graph.

Jeder Knoten des Graphen verwaltet eine Liste mit den bekannten Nachbarn. Diese enthält anfangs nur die initialen Nachbarn. In jeder Runde des FA stellt jeder Knoten des Netzes fest, welche neuen Nachbarn er seit der letzten Runde erhalten hat. Die Liste neuer Nachbarn sendet der

betreffende Knoten an alle seine initialen Nachbarn. Diese nehmen alle Verbindungen, die sie noch nicht kannten, in die List ihrer Nachbarn auf (es entsteht jeweils eine neue Kante).

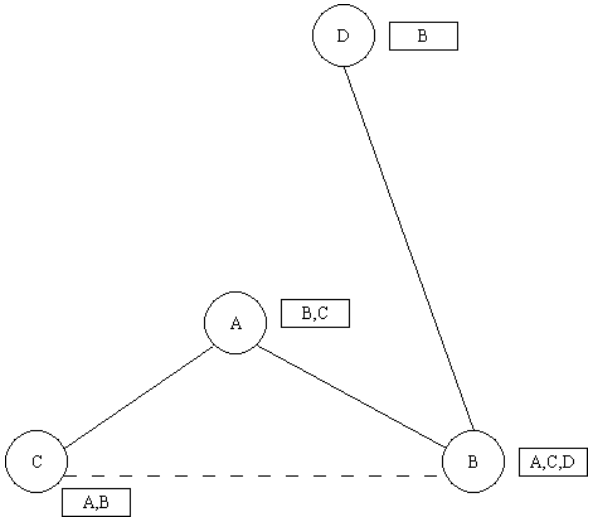


Abbildung 2

Runde 1: Erster Abgleich der Listen. Doppelte Halbkanten werden durch ungerichtete Kanten ersetzt. Neue Kanten werden unterbrochen dargestellt.

Die Anzahl der Runden, die der FA benötigt um einen vollständigen Graph zu erzeugen, entspricht dem Durchmesser (diameter) des initialen Graphen. Daher kann der FA sehr ineffizient arbeiten, falls er mit einem initialen Graph beginnt, der einen großen Durchmesser hat. Eine genauere Analyse des FA, sowie ein erweiterter Algorithmus findet sich in [7].

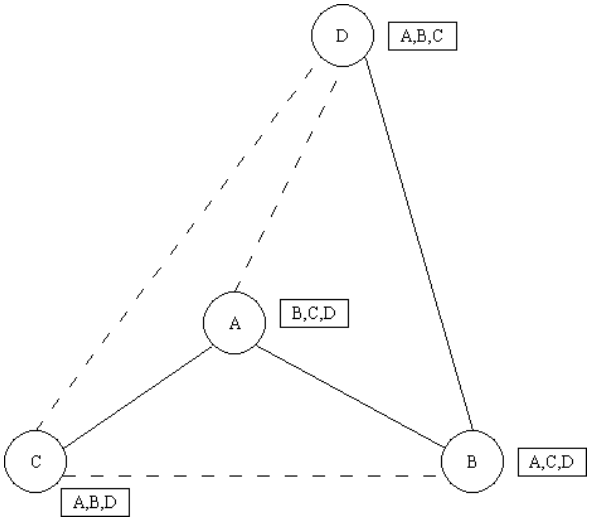


Abbildung 3

Runde 2: Abgleich der Listen abgeschlossen. Der Graph ist vollständig.

2.4 Peer-to-Peer

After a year or so of attempting to describe the revolution in file sharing and related technologies, we have finally settled on a label for what's happening: peer-to-peer.

-Clay Shirky, November 2000

Perhaps it is time for another major conceptual leap, where we let go of the notion of location. Welcome to the Heisenberg Principle as applied to the Internet.

-Andy Oram about Peer-to-Peer

Um zu beschreiben was Peer-to-Peer ausmacht, bedient man sich am besten der Worte von Clay Shirky [22]:

P2P is a class of applications that takes advantage of resources -- storage, cycles, content, human presence -- available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers.

Peer-to-Peer-Netze (*P2P*) [6] lassen sich bezüglich der Konnektivität in drei grobe Kategorien unterteilen: zentral verwaltet, hierarchisch und dezentralisiert.

In zentral verwalteten Systemen übernimmt ein einzelner Server die Koordination der einzelnen Teilnehmer. Ein solches System wird auch dann noch als zentral verwaltet betrachtet, wenn die Teilnehmer sich mit Hilfe der Informationen vom zentralen Server später selbst untereinander kontaktieren. Das ist beispielsweise bei *SETI@home* [14] und *Napster* [15] der Fall.

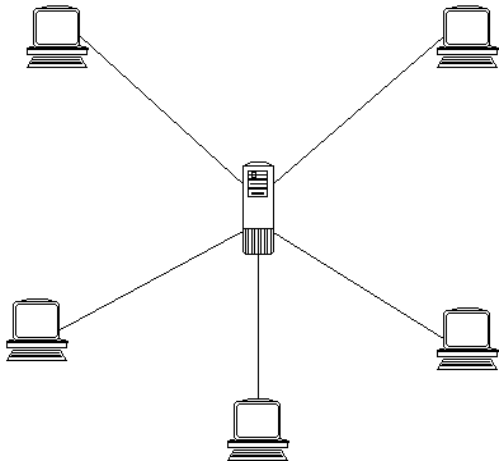


Abbildung 4

Ein zentral verwaltetes System

Ein hierarchiches System ist wie ein Baum strukturiert. Die Teilnehmer werden in eine Hierarchie von Gruppen eingeteilt, und ein lokaler Koordinator vermittelt die Kommunikation zwischen den Teilnehmern der gleichen Gruppe. Kommunikation zwischen Teilnehmern aus verschiedenen Gruppen wird an einen Koordinator übertragen, der in der Hierarchie höher angesiedelt ist. Das gleiche Prinzip wird bei dem *Domain Name System* (*DNS*) [17] verwendet.

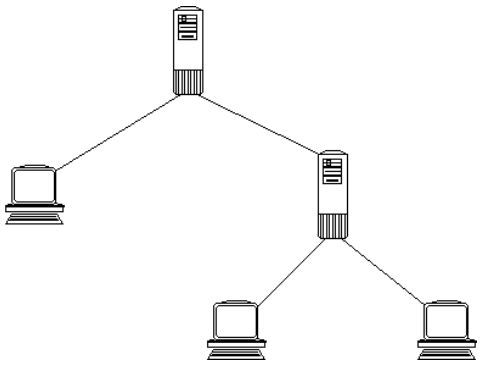


Abbildung 5

Ein hierarchisches System

Vollständig dezentralisierte Systeme haben gar keine globale Koordination. Die Kommunikation wird vollständig von den einzelnen Teilnehmern auf lokaler Ebene verwaltet. Dazu ist in der Regel ein Mechanismus zum Weiterleiten von Nachrichten mit Hilfe der anderen Teilnehmer notwendig. *Freenet* [13] und *Gnutella* [16] sind Beispiele aus dieser Kategorie.

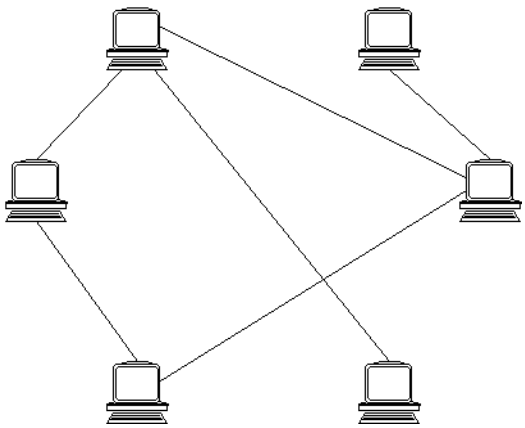


Abbildung 6

Ein dezentralisiertes System

Nun soll am Beispiel von Gnutella verdeutlicht werden wie sich die Mechanismen zum initialen Finden von anderen Gnutella-Teilnehmern,

also die Resource-Discovery, im Laufe der Zeit gewandelt haben.

In den Anfangstagen von Gnutella findet man seinen Weg per Mund-zu-Mund-Propaganda oder Host-Listen ins Netz. Man erhält beispielsweise via IRC oder von einer Website die Adresse eines bereits mit dem Gnutella-Netzwerk verbundenen Hosts. Diese gibt man in seinem Gnutella-Client ein, welcher den Rest der Arbeit übernimmt. Dieses Verfahren wird aber für den Benutzer sehr aufwendig, wenn die Anzahl der Teilnehmer stark zunimmt und aufgrund der hohen Dynamik der Konnektivität von Peer-to-Peer-Netzen viele Adressen ungültig, also nicht mehr verbunden sind.

Die Lösung für diese Problematik ist ein *Host-Cache*. Dieser dient als eine Art Sprungbrett ins Gnutella-Netz. Es handelt sich dabei um einen idealer Weise permanent laufenden Host (mit einer festen Adresse), der jedem neuen Teilnehmer automatisch eine aktuelle List der anderen Gnutella-Teilnehmer übergibt. Ein Teilnehmer braucht sich jetzt nur noch auf eine einzelne Adresse zu verlassen und erspart sich so einige Mühe.

Allerdings stellt diese Lösung wieder einen Schritt in Richtung Zentralisierung dar. Ein Host-Cache kann eine negative Wirkung auf die Selbstorganisationsfähigkeit des Netzwerkes haben. Problematisch wird ein Host-Cache in dem Moment, in dem er sehr populär wird. Dann gibt es nämlich eine große Anzahl von Teilnehmern, von denen jeder jeden anderen Teilnehmer kennt. Ein Host-Cache vermittelt nur Teilnehmer, die er vor einer realtiv kurzen Zeit „gesehen“ hat. Melden sich jetzt viele Teilnehmer am gleichen Host-Cache an, entsteht also ein stark gebündelter Cluster von Gnutella-Teilnehmern, die alle miteinander in Verbindung stehen. Das Resultat ist eine Überbelastung der Kommunikationswege und ein dadurch verursachtes Erstarren des Datenverkehrs. Ein Gnutella-Netz dieser Form lässt sich mit einem überfüllten Lokal vergleichen, in dem sehr viel geredet wird. Man kann sich zwar noch unterhalten, aber nur noch mit Personen die sich in unmittelbarer Umgebung befinden.

Ohne die Verwendung eines Host-Caches ist es stark dem Zufall überlassen, zu welchem Teil des Gnutella-Netzes man Verbindung aufnimmt. Fragt man zwei verschiedene Personen, nennen sie einem wahrscheinlich zwei verschiedene Adressen. Zwei verschiedene Host-Listen enthalten selten die gleichen Kontakte. Dadurch entstehen viele kleine Cluster mit lockeren Kommunikationsverbindungen. Ein Gnutella-

Netz dieser Form lässt sich mit einer Landkarte vergleichen, auf der sich viele kleine Städte und Ortschaften befinden, die mit nur wenigen Strassen untereinander verbunden sind. Dies scheint einer der Fälle zu sein, in denen sich eine kleine Ineffizienz positiv auswirkt.

Um die Situation mit den Host-Caches zu lösen, muss man diese entweder entfernen oder verbessern. Die Idee ist, dem Host-Cach so viel „Intelligenz“ zu verleihen, dass sie in der Lage sind, Teilnehmer sinnvoll auf bestehende Cluster zu verteilen und gegebenenfalls neue zu erstellen. Man könnte das vergleichen mit einem erfolgreichen Gastwirt, der ankommende Gäste in eine seiner anderen Kneipen schickt, sobald es in seinem Lokal zu laut wird. Problematisch ist hierbei, dass jeder Gnutella-Client einen lokalen *Host-Catcher* besitzt. In diesem Catcher wird eine lange Liste von allen Kontakten gespeichert, mit denen der Host in irgendeiner Form in Kontakt kam, sei es durch Anfragen, Antworten, Übertragungen etc. Diese Liste wird auch für alle weiteren Kontaktaufnahmen mit dem Gnutella-Netz verwendet, es werden dann keine Host-Listen oder Caches mehr benötigt. Dies hat eine gewisse Instabilität zur Folge, da Teilnehmer sich einfach mit Hosts verbinden, die sie kennen und dabei ignorieren, ob diese in Bezug auf Kapazität und Topolgie vielleicht eine schlechte Wahl sind. Die Gnutella-Clienten müssen also so modifiziert werden, dass sie Kontakte „vergessen“, die in irgendeiner Form ungeeignet sind. Mit Hilfe dieser Änderungen lässt sich die Leistung des Systems wieder verbessern ohne die Handhabung durch den Benutzer zu beeinträchtigen.

Weiter Details zu der Geschichte von Gnutella finden sich in [6].

2.5 Overlay-Netzwerke

Ein Overlay-Netzwerk entsteht aus einer Ansammlung von Knoten, die sich an bestimmten Positionen in einem bereits existierenden Netzwerk befinden. Diese Knoten bilden ein eigenes Netzwerk „über“ dem, zu Grunde liegenden *Substratnetzwerk*. Innerhalb eines solchen Overlays könnte man wieder bestimmte Knoten herraussuchen, die bestimmte Dienste anbieten. Auf diese Weise erhält man beliebige Schichten (*Layers*) von Netzwerken über Netzwerken.

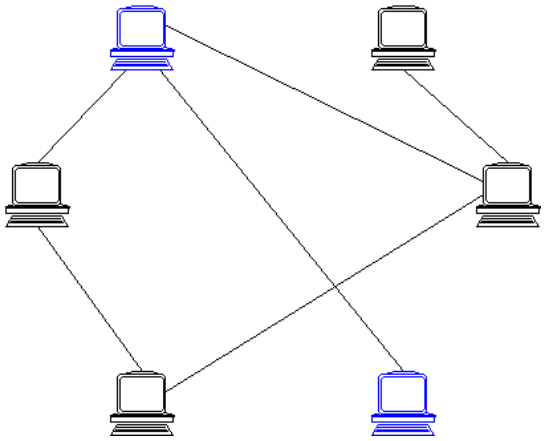


Abbildung 7

Die blauen Rechner bilden ein Overlay-Netzwerk über einem bestehenden Netz.

Die bereits genannten P2P-Systeme lassen sich als Overlay-Netzwerke bezeichnen. Das Internet stellt dann jeweils das Substratnetzwerk dar, und das Overlay-Netz selbst besteht aus allen Teilnehmern des P2P-Systems. Alle genannten Systeme haben ein wichtiges Problem gemeinsam: Wie findet man einen Kontaktknoten aus dem Overlay, dem man sich anschließen möchte, den initialer Kontakt? Es gibt mittlerweile eine zweite Generation von P2P-Systemen zu der u.a. Chord [18] und Pastry [19] gehören. Doch auch bei diesen bleibt das Problem bestehen. Im Fall von Pastry werden beispielsweise zwei Möglichkeiten genannt, um einen Kontaktknoten zu finden. Erstens, automatische Suche mit Hilfe von *expanding ring* IP-Multicast [21], einer Form von kontrolliertem *flooding*. Diese Methode ist jedoch nur dann effektiv, wenn ein relativ großer Anteil der Knoten im Internet zu dem gesuchten Overlay gehören. Zweitens, Bezug der Adresse über *outside channels*, was soviel heißt wie „Jemanden fragen, der sich damit auskennt“, z.B. einen Administrator oder eine bekannte Website. Eine dritte, naheliegende Möglichkeit ist, einem oder mehreren Knoten eine bekannte, feste IP-Adresse oder Domänennamen zuzuweisen.

In [20] wird die Problematik von Overlay-Systemen eingehender diskutiert. Es wird eine universelle Overlay-Infrastruktur vorgeschlagen. Die Methode von Pastry zum Finden des initialen Kontakts wird jedoch, im Prinzip, übernommen und nicht erweitert.

3 Analyse und Lösungsansatz

In diesem Kapitel werden die Resultate, die sich aus den Untersuchungen des vorherigen Kapitels ergeben, erläutert. Weiterhin wird das Konzept für die Lösung des Host-Discovery-Problems in SeMoA skizziert.

3.1 Die Henne und das Ei

Do we listen to pop music because we're depressed, or are we depressed because we listen to pop music?

-John, High Fidelity

Eine Ausprägung des klassischen Henne-Ei-Problems ist das Finden eines initialen Kontaktes, also eines Knotens, der zu einem Overlay-Netzwerk gehört, welchem man beitreten möchte. In den bisher untersuchten Anwendungen wurden zwar Mittel und Wege entdeckt dieses Ziel zu erreichen, diese sind aber jeweils nur in einem ganz bestimmten Szenario direkt anwendbar und besitzen auch unterschiedliche Vor- und Nachteile. Es scheint kein allgemeingültiges Verfahren zu existieren.

In den bekannten Methoden lassen sich drei unterschiedliche Ansätze unterscheiden:

1. Durchsuchung des relevanten Substratnetzes (*brute force*).
2. Netzfremde Speicherung und Bereitstellung von Kontaktadressen.
3. Feste Kontaktadressen für bestimmte Netwerkknoten.

Ansatz 1 wird in erster Linie durch Multicasting oder IP-Multicasting umgesetzt. Systeme, die damit arbeiten (können) sind beispielsweise Pastry, Chord und Vicinity. Problematisch ist hierbei die Effizienz von Multicasting in großen Netzwerken.

Ansatz 2 wurde am Beispiel von Gnutella vorgestellt (Host-Listen, IRC). Eine neue Frage, die sich hierbei ergibt ist: Wie und von wem werden diese Daten aktualisiert?

Ansatz 3 läßt sich Prinzipiell immer anwenden. Ein Beispiel hierfür wäre der Host-Cache von Gnutella. Die Schwierigkeiten, die sich dadurch für die Selbstorganisationsfähigkeit des Netzes ergeben, wurden bereits diskutiert. Ein anderer Punkt ist, daß Knoten mit festen Kontaktadressen in der Regel die Minderheit in einem Overlay-Netzwerk stellen, dadurch Gesamtsystem angreifbarer. Fallen hinreichen viele dieser Knoten aus, so ist es nicht mehr möglich auf diesem Wege neue Teilnehmer ins Netz zu integrieren. Man könnte hier auch von einer Form der Zentralisierung sprechen. Daß zentralisierte Systeme angreifbar sind, hat die Geschichte von Napster gezeigt.

3.2 SeMoA

Wie kann das Problem der Host-Discovery nun im Bezug auf SeMoA gelöst werden? Um diese Frage beantworten zu können, sollte man sich zunächst einmal zwei Dinge klar machen. Erstens, ist ein SeMoA-Netzwerk als Overlay auf dem Internet oder einem LAN zu verstehen. Daher lassen sich die vorgestellten Verfahren aus dem Gebiet der Overlay-Netzwerke bzw. P2P-Systeme hier anwenden. Zweitens, stellt SeMoA keine spezifische Applikation dar. SeMoA ist vielmehr als Infrastruktur zu interpretieren, auf deren Grundlage sich verschiedenste Anwendungen realisieren lassen. Daher muß eine Lösung, die Host-Discovery für SeMoA betreiben soll möglichst flexibel sein. Man kann nicht vorhersehen welche Anwendungen einmal auf der Grundlage von SeMoA arbeiten sollen, daher wäre es sehr schlecht wenn das, hier vorgestellte Modul gewisse Nachteile oder Einschränkungen aufweisen würde, die einen Einsatz in einem bestimmten Szenario unpraktikabel machen.

Zusammenfassend lassen sich folgende Kriterien nennen, die für den Entwurf von Farsight in Betracht gezogen wurden:

- Wiederverwendbarkeit
- Erweiterbarkeit
- Skalierbarkeit
- Arbeit auf Grundlage von Vicinity

Um Wiederverwendbarkeit zu fördern, wurde beschlossen den Dienst so einfach wie möglich zu gestalten. Es soll keine Topologie des Netzwerks verwaltet werden, sondern nur eine Liste von bekannten Kontaktadressen. Weiterhin soll es leicht möglich sein, die Kontakte zwischen verschiedenen

SeMoA-Systemen ausschließlich manuell zu konfigurieren.

Da es unterschiedliche Ansätze zu Host-Discovery gibt, sollen die verwendeten Verfahren austauschbar sein. Dafür bietet es sich an die Verfahren selbst in sogenannten *Discovery-Strategien* zu kapseln. Erweiterbarkeit bezieht sich vor allem auf den Austausch und das hinzufügen von Discovery-Strategien. Falls sich eine Strategie als ineffizient erweisen sollte, beispielsweise durch Änderungen am SeMoA-System, muss ein Austausch mit möglichst wenig Aufwand durchzuführen sein. Dabei sollten weder Farsight selbst noch andere Discovery-Strategien von Änderungen betroffen sein. Auch das Hinzufügen von ergänzenden Strategien sollte in diesem Sinne möglich sein.

Skalierbarkeit bezieht sich unter Anderem auf den Overhead bei der Kommunikation zwischen verschiedenen FarSight-Systemen. Mit Netzwerkresourcen soll möglichst schonend umgegangen werden. Der Nachrichtentransfer der durch Farsight entsteht sollte also möglichst minimal sein. Insbesondere sollte es zu keinem Lawinen-Effekt bei dem Versenden von Daten oder Agenten kommen, die ausschliesslich dem Zweck der Host-Discovery dienen.

Farsight soll Vicinity als Grundlage nutzen, und das in zweierlei Hinsicht. Zum Einen soll die lokale Host-Discovery innerhalb eines LAN Vicinity überlassen bleiben und Farsight soll lediglich auf diese Daten zugreifen. Zum Anderen soll die Struktur von Farsight an Vicinity angelehnt sein. Das heißt konkret, daß es einen lokalen Koordinator gibt, der die Daten von Farsight für andere Farsight-Dienste im LAN bereitstellt und aktualisiert. Nur dieser Koordinator soll für die Aktualisierung von bekannten Kontakten zuständig sein, die sich außerhalb des LAN befinden. Es sollen jedoch alle Teilnehmer in der Lage sein, neue Kontakte zu entdecken und dem Koordinator zu melden. Die Kontaktlisten aller Farsight-Dienste sollen außerdem stets synchronisiert werden. Falls der Koordinator ausfällt, übernimmt ein anderer Farsight-Dienst dessen Rolle. Optional soll es außerdem möglich sein jeden Farsight-Dienst unabhängig zu betreiben, also ohne synchronisation der Kontakte. Dieses Vorgehen wurde gewählt um alle drei anderen Kriterien, auf die ein oder andere Weise, zu unterstüzen.

Auf lokaler Ebene läßt sich folgende Struktur erkennen:

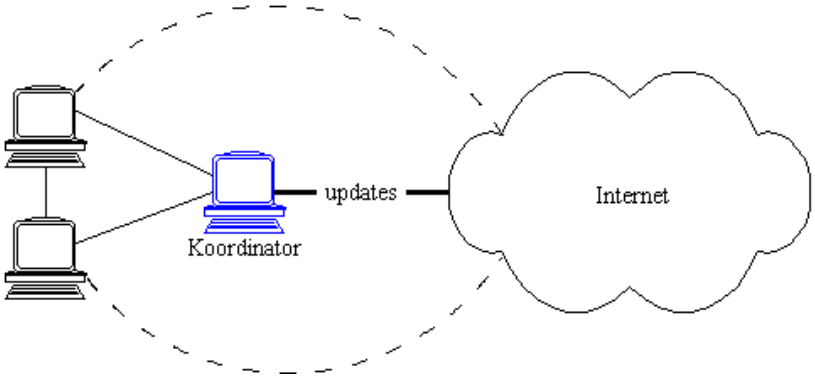


Abbildung 8

Der Farsight-Koordinator verwaltet und aktualisiert Kontakte innerhalb eines LAN.

Die exakte Funktionsweise von Farsight, sowie einer exemplarisch implementierten Discovery-Strategy wird im Kapitel *Implementierung* dargestellt.

4 Implementierung

Liberty means responsibility. That is why most men dread it.

-George Bernard Shaw

In diesem Kapitel soll die konkrete Implementierung des Farsight-Dienstes vorgestellt werden. Dabei wird sowohl die Installation und der Betrieb, als auch die zu Grunde gelegte Software-Architektur erläutert. Weiterhin werden die wichtigsten implementierungsspezifischen Details erklärt.

4.1 Überblick

Im Folgenden wird zunächst ein Überblick über die wichtigsten Eigenschaften der SeMoA-Plattform gewährt. Anschließend wird die Funktionsweise des Farsight-Dienstes zusammenfassend erläutert.

4.1.1 SeMoA

SeMoA steht für *Secure Mobile Agents*. Es handelt sich dabei um eine erweiterbare und offene Ablaufumgebung für mobile Agenten. Innerhalb eines Agentensystems führen Agenten geforderte Aufträge aus. Mobile Agenten sind weiterhin in der Lage das System in dem sie arbeiten zu verlassen und in einem anderen geeigneten Agentensystem ihre Arbeit fortzuführen. Man spricht dabei von *Migration*.

Der SeMoA-Server selbst ist in vollständig in Java implementiert und auch die Agenten können in Java geschrieben werden.

SeMoA konzentriert sich vor allem auf den Aspekt der Sicherheit mobiler Agenten, inklusive dem Schutz der Agenten vor so genannten *malicious hosts*.

Ein weiteres wichtiges Feature ist die Interoperabilität mit anderen Plattformen [10] wie *Aglets* und *JADE*. Agenten dieser Plattformen lassen sich in einem SeMoA-Server betreiben.

Innerhalb eines SeMoA-Systems lassen sich verschiedenste Dienste betreiben, die von Agenten benutzt werden können. Dienste lassen sich im so genannten *Environment* von SeMoA publizieren und können dort wieder abgerufen werden. Das Environment ist ein Namensraum, der global verfügbar im jeweiligen Server ist.

4.1.2 Farsight

In dieser Arbeit soll ein erweiterbarer Lösungsansatz zur Host-Discovery für SeMoA-Server implementiert werden. Farsight soll es ermöglichen, SeMoA-Server im Netz aufzuspüren, und diese in eine Kontaktliste aufzunehmen.

Für das LAN steht mit dem Vicinity-Modul bereits ein solcher Dienst zur Verfügung. Dieser arbeitet jedoch ausschließlich innerhalb der Grenzen eines LAN und ist damit nicht in der Lage, weiter „entfernte“ Server zu finden (z.B. im Internet).

Farsight stellt einen eigenständigen SeMoA-Dienst dar. Dieser arbeitet mit beliebigen und austauschbaren Sub-Modulen zusammen, die das eigentliche Auffinden von SeMoA-Servern betreiben. Farsight selbst hat die Aufgabe, die verschiedenen Suchergebnisse zu verwalten und zu pflegen, sowie die Liste von verfügbaren Servern über einen öffentliche Schnittstelle anderen Diensten und Agenten zur Verfügung zu stellen.

Das eigentliche Farsight-Modul lässt sich als ein Verzeichnisdienst betrachten. In die Kontaktliste von Farsight lassen sich auf drei unterschiedlichen Wegen neue Kontakte eintragen. Am naheliegendsten ist die direkte Eingabe einer Adresse zur Laufzeit durch den Benutzer. Dafür steht ein Visualisierungs-Plugin für SeMoA, zur Verfügung. Eine andere Möglichkeit stellt die Angabe von Kontakt-Adressen über die Farsight-Konfigurationsdatei dar. Diese werden beim Start von Farsight verarbeitet. Der wichtigste und interessanteste Weg ist jedoch das automatische Aufspüren von SeMoA-Servern. Dafür stellt Farsight eine öffentliche Schnittstelle zur Verfügung, über die andere SeMoA-Dienste oder Agenten Kontakte bekannt machen können. Die eigentliche Arbeit übernimmt in diesem Szenario eine so genannte *Discovery-Strategie*. Nach Eingabe einer Adresse prüft Farsight stets, ob der Kontakt dort tatsächlich zu erreichen ist. Falls ja, wird er unter dem Namen (*Distinguished Name*) des Zertifikates des SeMoA-Betreibers in die Kontaktliste aufgenommen.

Farsight lässt sich in zwei unterschiedlichen Modi betreiben. Dem *Shared-*

Mode und dem *Private-Mode*. Im Shared-Mode werden die Kontaktlisten zwischen allen Farsight-Diensten innerhalb des selben LAN, die auch im Shared-Mode arbeiten, synchronisiert. Im Private-Mode arbeitet ein Farsight-Dienst unabhängig von den anderen im LAN. Er gibt seine Kontakte nicht weiter und ignoriert auch neu entdeckte Kontakte von den anderen Farsight-Diensten aus dem gleichen LAN.

4.1.3 Discovery-Strategien

Eine Discovery-Strategie ist ein Sub-Modul von Farsight, welches das automatische Entdecken von SeMoA-Servern übernimmt. Die Architektur sieht vor, dass mehrere solcher Module parallel betrieben werden können.

Zur Zeit steht in Farsight eine einzige Discovery-Strategie, die *Explorer-Strategie*, zur Verfügung. Diese Strategie bedient sich mobiler Agenten, den *Explorer-Agents*, welche das Auskunschaften von fernen Netzen und den Transport von Kontaktlisten übernehmen.

Das Modul beginnt seine Tätigkeit in dem Moment, wenn ein Agent von Außerhalb das lokale SeMoA-System betritt. Es wird überprüft, woher der ankommende Agent stammt. Ist diese Kontakt-Adresse noch nicht bekannt, wird ein Explorer-Agent entsendet, um die Adresse zu erforschen.

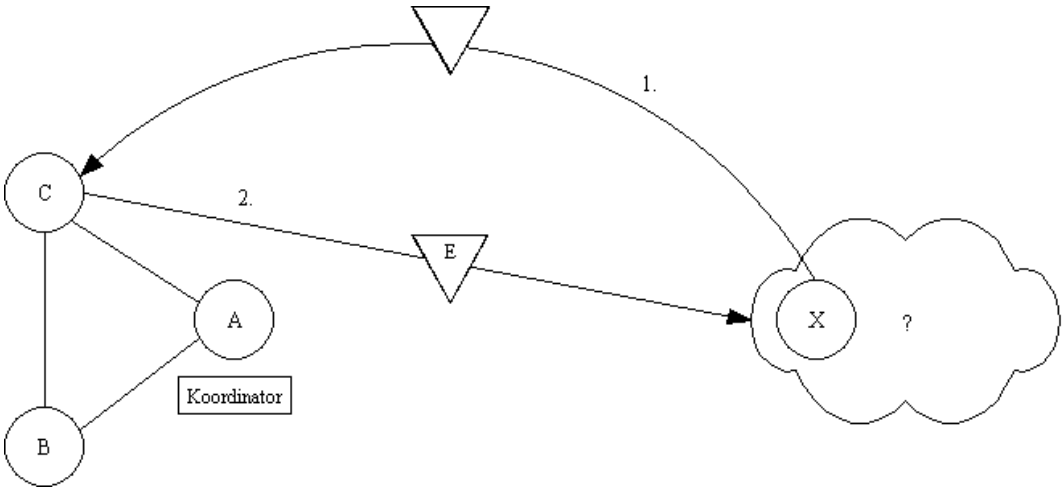


Abbildung 9

1.: Ein Agent migriert von X nach C.

2.: Ein Explorer-Agent wird nach X gesandt. Er enthält alle Kontakte des lokalen Netzes.

Ist der Agent am Ziel angekommen, so wird er dort außerdem versuchen Informationen über das LAN zu sammeln.

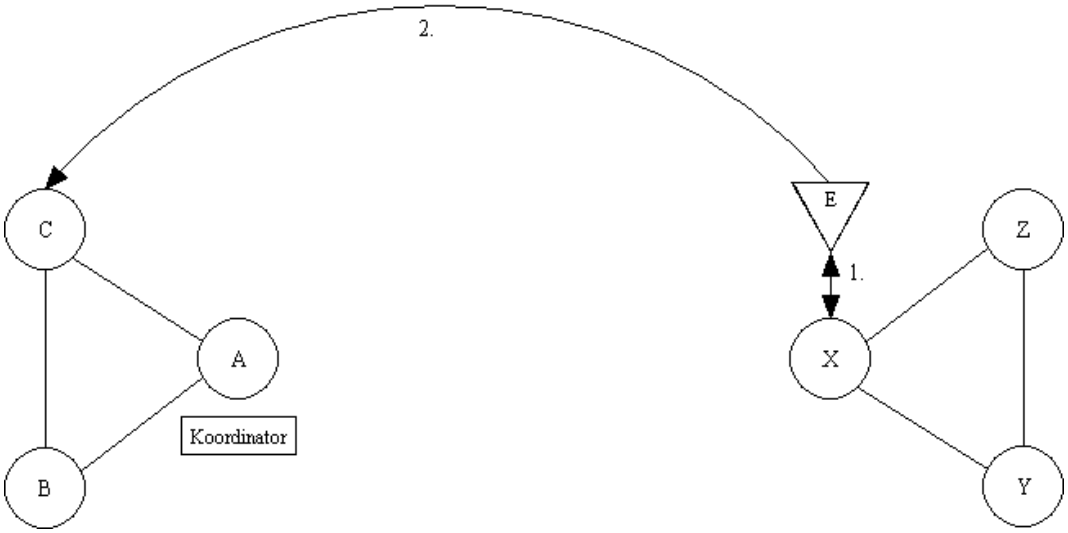


Abbildung 10

1.: Der Explorer-Agent ist im unbekannten Netz, auf dem System X eingetroffen. Die mitgebrachten Kontakte werden übergeben und die lokalen Kontakte geladen.

2.: Der Agent springt zurück ins Heimatsystem.

Ist der Agent zurückgekehrt sind dem lokalen SeMoA-Netz alle Kontakte aus dem fremden Netz bekannt.

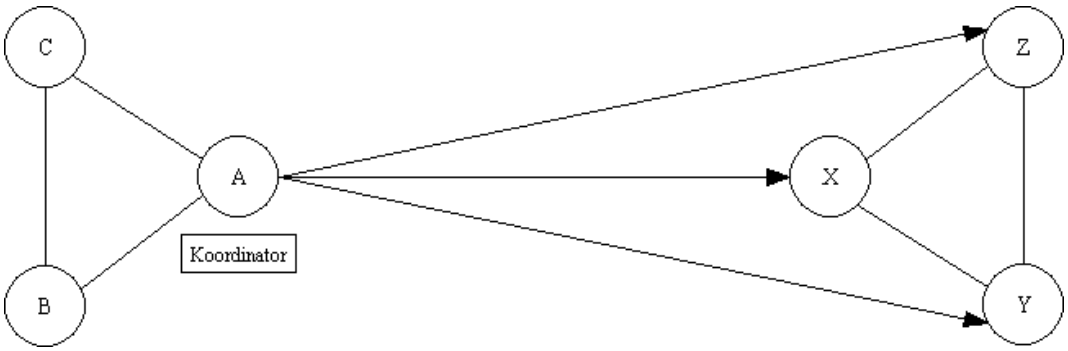


Abbildung 11

Die Kontakte sind ausgetauscht. Alle Systeme im Heimatnetz kennen alle Systeme im Zielnetz und umgekehrt. Der Koordinator übernimmt jeweils die Aktualisierung der Kontakte. (Zur Vereinfachung wurde dies nur aus Sicht des Heimatnetzes dargestellt)

4.2 Architektur

Im Folgenden wird die Architektur der Farsight-Implementierung vorgestellt. Es erfolgt ein Überblick über die einzelnen Software-Komponenten und deren Zusammenspiel, sowohl untereinander, als auch mit SeMoA und anderen SeMoA-Modulen.

Zunächst einmal einen grobe Einordnug von Farsight in die SeMoA-Umgebung. Die für Farsight wichtigen SeMoA-Module sind Vicinity und Ship. Für die Explorer-Strategie spielt weiterhin das In-Gate ein wichtige Rolle.

Vicinity ist ein automatischer Host-Discovery-Dienst. Er bietet Informationen über die SeMoA-Server, die sich im LAN befinden.

Ship steht für *simple host information protocols*. Über diesen Dienst lassen sich Informationen über entfernte SeMoA-Plattformen einholen.

Das In-Gate ist eine Kernkomponente des SeMoA-Systems. Es handelt sich dabei um eine Art Pipeline, in der sich so genannte Filter installieren lassen. Jeder Agent, der das System betreten will, durchläuft diese Pipeline und wird von den Filtern entsprechend bearbeitet.

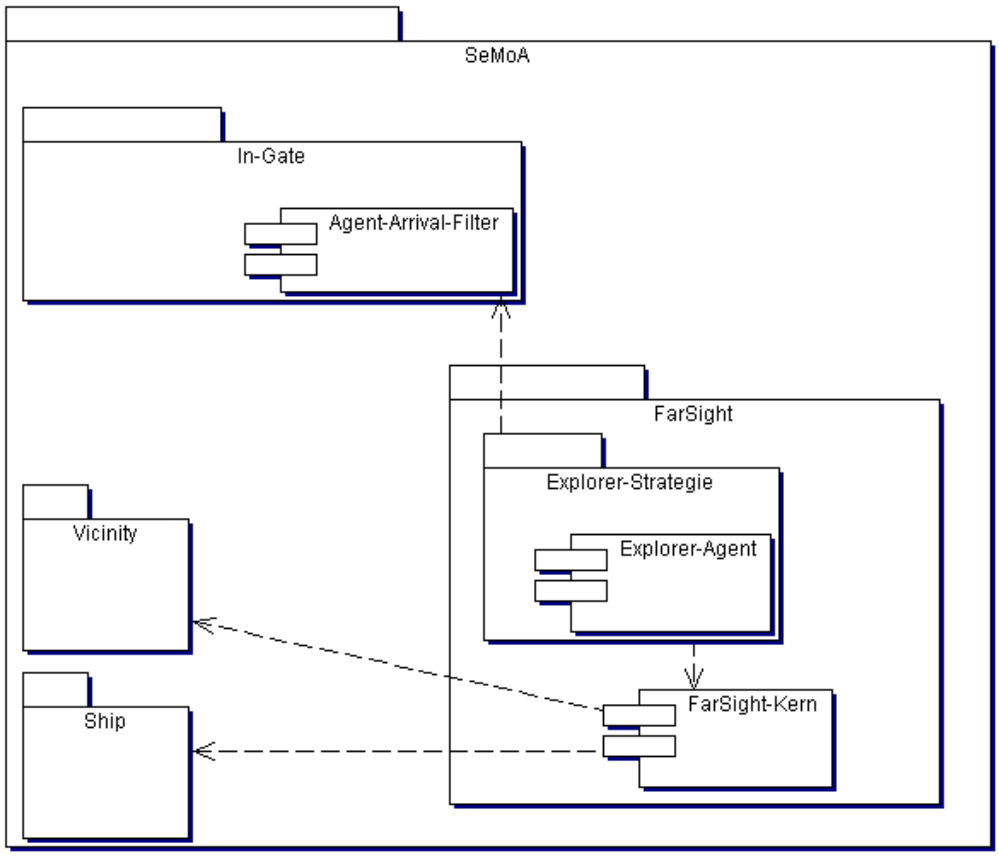


Abbildung 12

UML-Komponentendiagramm: Einordnung von Farsight in SeMoA.

Farsight bedient sich des Ship-Dienstes, um Anfragen an andere SeMoA-Server zu senden. Dabei interessiert vor allem, ob ein Kontakt überhaupt erreicht werden kann, und wer ihn betreibt. Vicinity wird benutzt, um zu klären ob ein bestimmter Kontakt aus dem eigenen LAN stammt. Farsight kennt auch wirklich nur diese beiden Module, und weiss nichts über vorhandene Discovery-Strategien.

Die Explorer-Strategie installiert bei ihrer Initialisierung den Agent-Arrival-Filter im In-Gate. Dieser reicht ankommende Agenten an die Strategie weiter, so dass diese näher inspiziert werden können. Werden der Strategie neue Kontakte bekannt, so werden diese an Farsight weitergeleitet.

Nun zum inneren Aufbau des Farsight-Dienstes. Farsight bedient sich zweier Sub-Komponenten, die jeweils in einem eigenen Prozess arbeiten.

Nämlich dem Update-Daemon und dem Poll-Daemon.

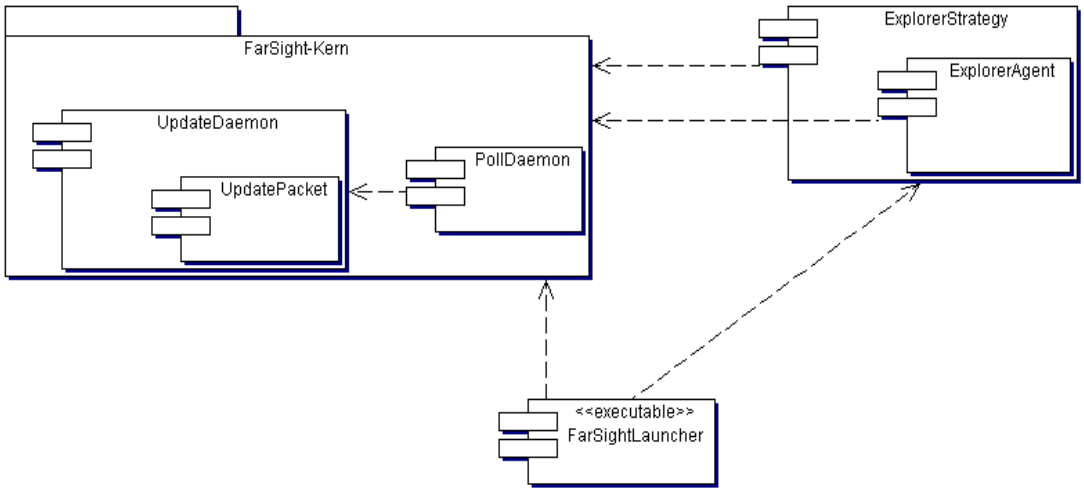


Abbildung 13

UML-Komponentendiagramm: Abhängigkeiten der internen Farsight-Komponenten.

Der Update-Daemon hat die Aufgabe die Kontaktlisten verschiedener Farsight-Dienste innerhalb eines LAN zu synchronisieren. Dabei werden so genannte Update-Pakete via IP-Multicasting verschickt. Diese Pakete enthalten Informationen über einen bestimmten Kontakt.

Der Poll-Daemon testet die Erreichbarkeit von bekannten Kontakten. Dies geschieht mit Hilfe des Ship-Dienstes. Ist ein Kontakt nicht mehr erreichbar, so wird dies dem Update-Daemon mitgeteilt, welcher die anderen Farsight-Dienste informiert.

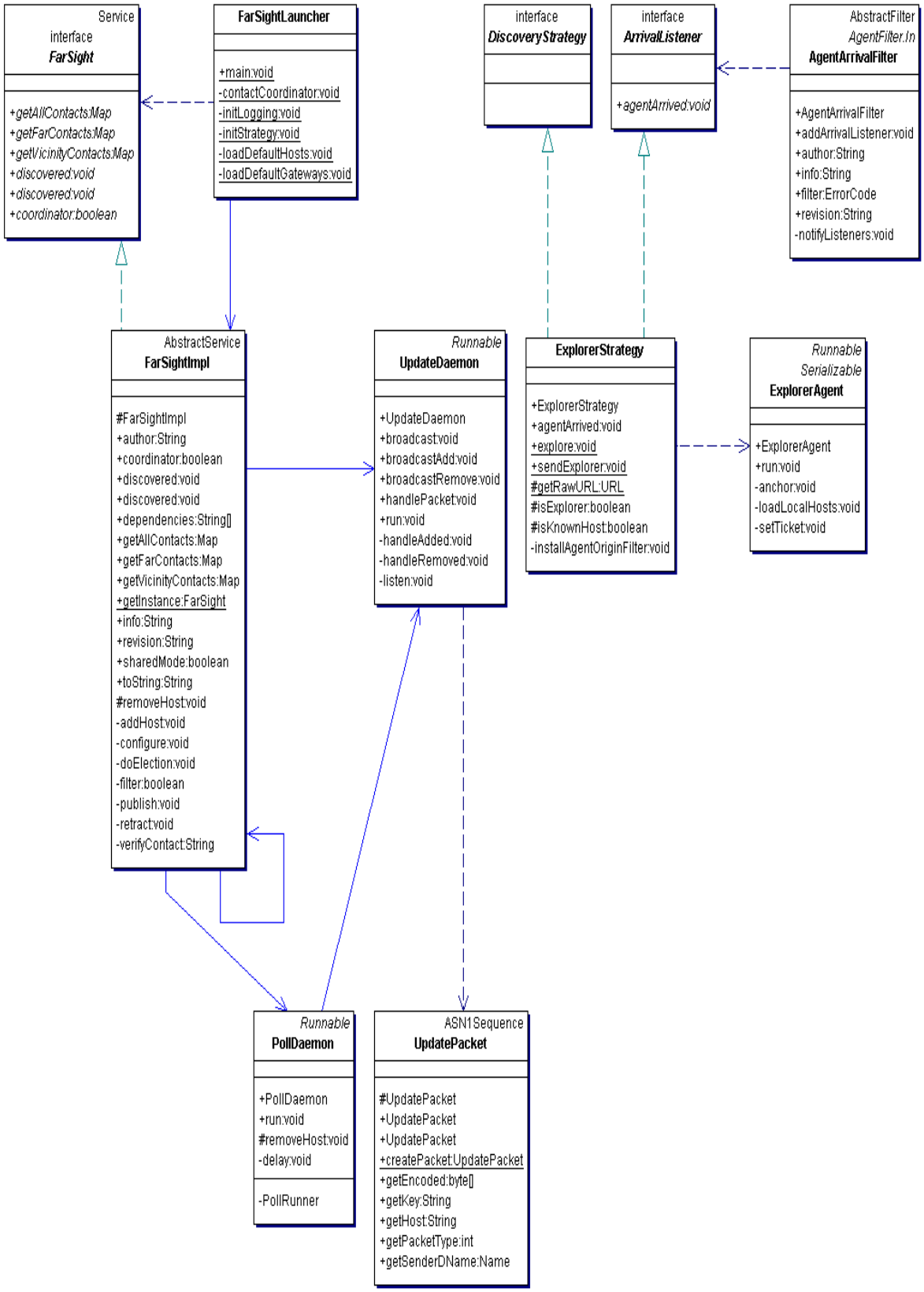
Der Farsight-Launcher ist für die Initialisierung des Gesamtsystems zuständig. Er liest die Konfigurationsdatei aus und initialisiert Farsight entsprechend. Ausserdem werden hier die Discovery-Strategien gestartet.

4.3 Implementierungsdetails

Im Folgenden Abschnitt werden die einzelnen Java-Klassen vorgestellt und einige spezifische Implementierungsdetails diskutiert.

Klassenhierarchie

Das folgende UML-Klassendiagramm soll die Zusammenhänge zwischen den einzelnen Komponenten verdeutlichen, und einen Überblick über die bereitgestellten Funktionen verschaffen.


Service
interface
FarSight
+getAllContacts:Map
+getFarContacts:Map
+getVicinityContacts:Map
+discovered:void
+discovered:void
+coordinator:boolean
FarSightLauncher
+main:void
-contactCoordinator:void
-initLogging:void
-initStrategy:void
-loadDefaultHosts:void
-loadDefaultGateways:void
interface
DiscoveryStrategy
interface
ArrivalListener
+agentArrived:void
AbstractFilter
AgentFilter.In
AgentArrivalFilter
+AgentArrivalFilter
+addArrivalListener:void
+author:String
+info:String
+filter:ErrorCode
+revision:String
-notifyListeners:void
AbstractService
FarSightImpl
#FarSightImpl
+author:String
+coordinator:boolean
+discovered:void
+discovered:void
+dependencies:String[]
+getAllContacts:Map
+getFarContacts:Map
+getVicinityContacts:Map
+getInstance:FarSight
+info:String
+revision:String
+sharedMode:boolean
+toString:String
#removeHost:void
-addHost:void
-configure:void
-doElection:void
-filter:boolean
-publish:void
-retract:void
-verifyContact:String
Runnable
UpdateDaemon
+UpdateDaemon
+broadcast:void
+broadcastAdd:void
+broadcastRemove:void
+handlePacket:void
+run:void
-handleAdded:void
-handleRemoved:void
-listen:void
ExplorerStrategy
+ExplorerStrategy
+agentArrived:void
+explore:void
+sendExplorer:void
#getRawURL:URL
#isExplorer:boolean
#isKnownHost:boolean
-installAgentOriginFilter:void
Runnable
Serializable
ExplorerAgent
+ExplorerAgent
+run:void
-anchor:void
-loadLocalHosts:void
-setTicket:void
Runnable
PollDaemon
+PollDaemon
+run:void
#removeHost:void
-delay:void
-PollRunner
ASN1Sequence
UpdatePacket
#UpdatePacket
+UpdatePacket
+UpdatePacket
+createPacket:UpdatePacket
+getEncoded:byte[]
+getKey:String
+getHost:String
+getPacketType:int
+getSenderDName:Name

AgentArrivalFilter

Diese Klasse implementiert den Agent-Arrival-Filter. An dem Filter lassen sich beliebige ArrivalListener registrieren. Ist der Filter in SeMoAs In-Gate installiert, leitet er alle ankommenden Agenten an die registrierten ArrivalListener weiter. Diese Klasse stellt zusammen mit dem ArrivalListener-Interface eine Implementierung eines leicht modifizierten Observer-Patterns [4] dar.

Vom SeMoA-System wird die Methode *filter* aufgerufen. Von dort aus wird für jeden ArrivalListener ein eigener Prozess erzeugt, der den Agenten übergibt. Die Idee dahinter ist, dass der Prozess der Filter-Pipeline wieder seine Arbeit aufnehmen kann, unabhängig davon was in den einzelnen ArrivalListenern passiert.

ExplorerStrategy

Diese Klasse implementiert zusammen mit dem ExplorerAgent die Discovery-Strategie *Explorer-Strategie*. ExplorerStrategy implementiert das ArrivalListener-Interface und registriert sich beim Agent-Arrival-Filter. Ein ankommender Agent wird von der Strategie auf bestimmte Eigenschaften überprüft. Wird der Agent akzeptiert, so wird ein Explorer-Agent erstellt, der den Server untersucht von dem der Agent zuletzt kam.

Folgendes darf auf den ankommenden Agenten **nicht** zutreffen:

- Ist selbst ein ExplorerAgent
- Herkunft unbekannt
- Herkunft liegt im LAN (ist bei Vicinity bekannt)
- Herkunft ist Farsight bereits bekannt

Trifft all dies nicht zu, übernimmt der ExplorerAgent den Rest der Arbeit.

ExplorerStrategy bietet weiterhin anderen Diensten oder Agenten die Möglichkeit mit Hilfe der statischen Methode *sendExplorer* (oder *explore*) zu beliebigen Zielen Explorer-Agenten zu senden.

Im der Konfigurationsdatei von Farsight lässt sich eine Liste von *DefaultGateways* angeben, die von der Explorer-Strategie beim Systemstart verarbeitet werden. An jeden Kontakt aus dieser Liste wird ein Explorer-Agent gesendet.

ExplorerAgent

Diese Klasse implementiert den Explorer-Agenten. Seine Aufgabe ist es im Rahmen der Explorer-Strategie SeMoA-Kontakte zu sammeln und zwischen verschiedenen Netzen zu transportieren. Dabei migriert er genau zwischen zwei Adressen: Einem beliebigen Ziel und seinem Heimatsystem.

Der Agent arbeitet wie folgt:

1. Alle Kontakte die Farsight und Vicinity auf dem lokalen SeMoA-System bekannt sind werden gespeichert
2. Der Agent springt zu seiner Ziel-Adresse
3. Die gespeicherten Kontakte werden dem lokalen Farsight übergeben
4. Alle Kontakte die Farsight und Vicinity auf dem lokalen SeMoA-System bekannt sind werden gespeichert
5. Der Agent springt zu seiner Heim-Adresse
6. Die gespeicherten Kontakte werden dem lokalen Farsight übergeben

Das sehr einfache Verhalten des Agenten ist darauf zurückzuführen, dass in SeMoA-Netzen eine gewisse Hierarchie besteht. Lokale SeMoA-Systeme, also solche die sich im gleichen LAN befinden, sind über den Dienst Vicinity mit einander verbunden. Der Explorer-Agent stellt nur noch eine Verbindung zwischen diesen Sub-Netzen her.

PollDaemon

Der Poll-Daemon testet die Erreichbarkeit von bekannten Kontakten. Er arbeitet direkt auf der Kontaktliste von Farsight. In periodischen Zeitabständen (*PollDelay*) wird zufällig ein Kontakt ausgewählt und daraufhin überprüft, ob dieser noch erreichbar ist. Dies geschieht jedoch nur, solange das lokale Farsight auch der Koordinator im LAN ist.

Die Anfrage geschieht mit Hilfe des Ship-Dienstes (*ship.isAlive*). Fällt die Anfrage negativ aus, so wird der Kontakt aus Farsights Kontaktliste entfernt, und über den Update-Daemon werden die anderen Farsight-Dienste im LAN über den Ausfall des Kontakts informiert. Dies alles geschieht in einem eigenen Prozess, da eine einzelne Ship-Anfrage sehr lange dauern kann. Die maximale Anzahl der generierten Prozesse wird über einen *ThreadPool* gesteuert.

Diese Klasse implementiert das Self-Running-Object-Pattern [5].

UpdateDaemon

Der Update-Daemon hat die Aufgabe, die Kontaktlisten verschiedener Farsight-Dienste innerhalb eines LAN zu synchronisieren. Er arbeitet nur, falls Farsight im SharedData-Mode gestartet wurde. Während der Update-Daemon läuft, lauscht er auf einer IP-Multicasting-Adresse nach Update-Paketen (*UpdatePacket*). Ein Update-Paket enthält den Typ des Updates (*Add* oder *Remove),* die System-Id des SeMoA-Systems, welches das Paket sendet, und die System-Id und Ship-Adresse des SeMoA-Systems dessen Status sich geändert hat. Der Update-Daemon besitzt eine Referenz der Kontaktliste von Farsight, und kann entsprechend einen neuen Kontakt hinzufügen oder einen veralteten löschen.

Über die öffentliche Broadcasting-Schnittstelle (*broadcast, broadcastAdd, broadcastRemove*) besteht für andere Komponenten die Möglichkeit, Update-Pakete zu senden (z.B. Poll-Daemon oder Farsight).

Diese Klasse implementiert das Self-Running-Object-Pattern [5].

FarSightLauncher

Der Farsight-Launcher ist für die Initialisierung des Gesamtsystems zuständig. Hier wird die Konfigurationsdatei von Farsight geladen und bearbeitet. Die Position der Konfigurationsdatei muss beim Start der Main-Methode als Parameter übergeben werden. Der Launcher übernimmt folgende Aufgaben:

- Starten von Farsight (*FarSightImpl*)
- Starten der Discovery-Strategien
- Bekanntmachen der *DefaultHosts* bei Farsight

FarSightImpl

Diese Klasse implementiert das Farsight-Interface. Farsight arbeitet ähnlich wie ein Verzeichnisdienst. Grundsätzlich lassen sich Kontakte über eine öffentliche Schnittstelle eingeben *(discovered)* und wieder abfragen *(getAllContacts, getFarContacts, getVicinityContacts).* Die Kontakte werden in einer *Map* gespeichert, dabei dient die Id (*ownerDN*) des Kontakts als Schlüssel, und die Adresse (Ship-Adresse) als Wert.

Farsight lässt sich in zwei unterschiedlichen Modi betreiben. Dem *Shared-Mode* und dem *Private-Mode*.

Arbeitet Farsight im Private-Mode, so werden andere Farsight-Dienste im

LAN ignoriert. Das hat zur Folge, dass dieser Dienst alle seine bekannten Kontakte selbst mit Hilfe des Poll-Daemon, in periodischen Abständen daraufhin überprüft ob diese noch erreichbar sind. Außerdem werden neu entdeckte Kontakte, sowie nicht mehr erreichbare Kontakte, nicht mehr an die anderen Farsight-Dienste im LAN via Update-Daemon weiter gemeldet.

Arbeitet Farsight im Shared-Mode, so wird innerhalb des LAN ein Farsight-Koordinator gewählt. Dies geschieht mit Hilfe des Voting-Dienstes, den Vicinity anbietet. Der Koordinator übernimmt die Überprüfung der bekannten Kontakte, und meldet nicht erreichbare Kontakte via Update-Daemon an die anderen Farsight-Dienste im LAN weiter. Alle anderen Farsight-Dienste innerhalb des gleichen LAN, die auch im Shared-Mode arbeiten, lauschen ob Nachrichten eintreffen, und senden ihrerseits neu entdeckte Kontakte an die anderen Dienste (Update-Daemon).

Insgesamt kann Farsight bezüglich dieses Parameters also in drei verschiedenen Zuständen arbeiten:

- Private-Mode
- Shared-Mode und Koordinator
- SharedMode und kein Koordinator

Bei der Eingabe wird ein Kontakt auf seine Gültigkeit überprüft. Ein Kontakt ist dann gültig, wenn er Vicinity nicht bekannt ist und durch eine Anfrage des Ship-Dienstes erreicht werden kann (*ship.isAlive*). Die Anfrage läuft in einem eigenen Prozess, da eine einzelne Ship-Anfrage unter Umständen recht viel Zeit in Anspruch nehmen kann. Die maximale Anzahl der generierten Prozesse wird über einen *ThreadPool* gesteuert. Fällt die Anfrage positiv aus, so wird via Ship die Id (*ownerDN*) des Hosts erfragt. Anschliessend wird unter der Id die Ship-Adresse des Kontakts in Farsight gespeichert. Arbeitet Farsight im Shared-Mode werden ausserdem die anderen Farsight-Dienste im LAN mit Hilfe des Update-Daemon benachrichtigt.

Diese Klasse implementiert das Singelton-Pattern [4].

5 Installation und Betrieb

Dieser Abschnitt beschreibt alle Schritte, die für eine erfolgreiche Inbetriebnahme Farsights erforderlich sind.

5.1 Installation

Voraussetzung für eine Installation von Farsight ist eine vorhandene und betriebsfähige SeMoA-Umgebung. Näheres dazu findet sich in der Dokumentation von SeMoA [2]. Im Folgenden bezeichnet `$SEMOA` das Verzeichnis in dem SeMoA installiert wurde.

Die Installation von Farsight lässt sich in zwei Punkten zusammenfassen:

1. Einfügen des Farsight-Packages in die Package-Struktur von SeMoA
2. Anlegen der Konfigurationsdateien und Verzeichnisse

Zu 1.: Das Farsight-Package ist ein Verzeichnis mit dem Namen `farsight`, in dem sich alle benötigten Klassen befinden. Dieses Verzeichnis muss in die Verzeichnisstruktur von SeMoA eingefügt werden. Und zwar unter: `$SEMOA/classes/DE/FhG/IGD/semoa`.

Zu 2.: Unter `$SEMOA/etc` ist ein Verzeichnis mit dem Namen `farsight` zu erstellen. Dort sollte sich die Konfigurationsdatei mit dem Namen `farsight.conf` befinden. Auf den Inhalt dieser Datei wird im nächsten Punkt dieses Abschnitts eingegangen. Sollte die Datei nicht vorhanden sein, wird Farsight beim Start mit Standartparametern initialisiert.

Grundsätzlich kann sich die Konfigurationsdatei überall befinden, da ihre Position ohnehin beim Start von Farsight als Parameter übergeben werden muss (siehe Abschnitt *Starten*).

5.2 Konfiguration

Die Farsight-Konfigurationsdatei ist im *Properties* Format gehalten [3]. Es gibt vier verschiedene Kategorien von Parametern:

1. Der Farsight-Kern
2. Die Discovery-Strategien
3. Die Default-Hosts
4. Die Default-Gateways

Eine Beispieldatei

```
########################################
# The Farsight core configuration.
#
SharedDataMode          = on
LoadDefaultHosts        = off
PollDelay               = 5000
PollThreadPoolSize      = 20
DiscoveryThreadPoolSize = 20
MulticastPort           = 40000
MulticastAddress        = 224.0.24.0

#
# The default hosts.
#
DefaultHost.1 = ship://192.168.0.1:47470
#DefaultHost.2 = ship://192.168.0.3:47470

#
# The discovery strategy modules.
#
DiscoveryStrategy.1 = DE.FhG.IGD.semoa.farsight.ExplorerStrategy


########################################
# The Explorer Strategy
#
LoadDefaultGateways = off

#
# The default gateways.
#
DefaultGateway.1 = ship://192.168.0.1:47470
```

Im Folgenden sollen die einzelnen Parameter erläutert werden.

SharedDataMode

Dieser Parameter legt fest, ob Farsight sich mit anderen Farsight-Diensten innerhalb des LAN koordinieren und seine Kontaktlisten synchronisieren soll.

LoadDefaultHosts

Dieser Parameter gibt an, ob beim Start von Farsight die Liste der Default-

Hosts verarbeitet (`on`) oder ignoriert (`off`) werden soll.

LoadDefaultGateways

Dieser Parameter gibt an, ob beim Start von Farsight die Liste der Default-Gateways verarbeitet (`on`) oder ignoriert (`off`) werden soll.

PollDelay

Dieser Parameter gibt an in welchem Zeitintervall ein Kontakt aus der Kontaktliste daraufhin überprüft werden soll ob er noch erreichbar ist. Der Wert wird als Angabe in Millisekunden interpretiert.

PollThreadPoolSize

Die Anfragen des PollDaemon werden in eigenen Prozessen (*Threads*) verarbeitet. Dieser Parameter gibt an wie viele Prozesse maximal zu einer bestimmten Zeit aktiv sein sollen.

DiscoveryThreadPoolSize

Der Vorgang des Hinzufügens von neuen Kontakten in Farsight wird in einem eigenen Prozess (*Thread*) verarbeitet. Dieser Parameter gibt an wie viele Prozesse maximal zu einer bestimmten Zeit aktiv sein sollen.

MulticastAddress

Dieser Parameter bestimmt die IP-Adresse, die für den Austausch von Nachrichten zwischen den Farsight-Diensten innerhalb eines LAN verwendet werden soll.

MulticastPort

Dieser Parameter bestimmt den Port, der für den Austausch von Nachrichten zwischen den Farsight-Diensten innerhalb eines LAN verwendet werden soll.

DefaultGateway

Dieser Parameter gibt einen Kontakt an, zu dem beim Start von Farsight ein Explorer-Agent geschickt wird.

DefaultHost

Dieser Parameter gibt einen Kontakt an, der sofort beim Start von Farsight in die Kontaktliste aufgenommen werden soll.

DiscoveryStrategy

Dieser Parameter gibt ein Modul zur Host-Discovery an, das zusammen mit Farsight gestartet werden soll.

Der angegebene Wert wird als vollständiger Klassenname interpretiert. Das Modul wird mit dem Class-Loading-Mechanismus (*Class.for(Name)*) des *Reflection-Toolkits* [3] initialisiert.

5.3 Starten

Sollte Farsight beim Start von SeMoA noch nicht automatisch gestartet worden sein, so kann dies nachträglich über die SeMoA-Shell geschehen. Dabei wird das „java“ Kommando verwendet, dem als Parameter der Klassenname des Farsight-Startprogramms, sowie die relative Position der Kofigurationsdatei angeben werden müssen.

Das sieht beispielsweise so aus:

```
java DE.FhG.IGD.semoa.farsight.FarSightLauncher etc/farsight/farsight.conf
```

6 Schlussfolgerungen

Mit dem Farsight-Modul wurde ein flexibler und erweiterbarer Ansatz für die Lösung des Host-Discovery-Problems in SeMoA-Netzen vorgestellt.

Grundsätzlich lässt sich eine beliebige Netzwerktopologie manuell einrichten. Dafür eignen sich die DefaultGateway- und DefaultHost-Einträge der Farsight-Konfigurationsdatei. Für beliebige andere Ansätze lässt sich eine geeignete Discovery-Strategie nachträglich ins System einbetten.

Die vorgestellte Explorer-Strategie ist eher als eine Näherungslösung oder „Proof-of-Concept“ Implementierung zu verstehen. Die Strategie garantiert nicht das Auffinden aller, im Netz theoretisch erreichbaren, SeMoA-Server. Der Grund dafür ist, dass der Discovery-Vorgang stark an das Mirgationsverhalten der Agenten bzw. Dienste gekoppelt ist, die in den jeweiligen Servern eingesetzt werden. Es wird also kein vollständiger Graph modelliert. Weiterhin garantiert die Strategie keine Korrektheit. Subnetz-Informationen werden nach einem initialen Abgleich nicht aktualisiert. In einem eigentlich bekannten, entfernten SeMoA-Netz können nach dem initialen Kontakt neue Server hinzukommen, ohne dass das lokale explizit Netz davon erfährt. Nur bekannte Kontakte werden daraufhin überprüft, ob sie noch erreichbar sind.

Im Prinzip wurde das Henne-Ei-Problem durch die Explorer-Strategie nicht gelöst. Damit die Explorer-Strategie arbeiten kann müssen fremde Agenten das lokale System betreten. Damit sie das tun können, müssen sie wissen unter welcher Adresse das lokale System zu erreichen ist. Womit wir wieder beim Ausgangspunkt angelangt sind: Woher kennt der Agent diese Adresse? Der Einsatz der Explorer-Strategie macht also nur dann Sinn, wenn zuvor ein Teil des gewünschten Netzwerkes manuell konfiguriert worden ist.

Es ist außerdem unklar, ob Farsight tatsächlich skalierbar ist. Bisher wurden keine Simulationen oder Feldtests durchgeführt um das zu überprüfen.

7 Ausblick

Abschließend sollen noch einige Punkte, die die Weiterentwicklung von Farsight betreffen, diskutiert werden.

Der Punkt Sicherheit wurde bei der vorliegenden Implementierung weitestgehend außer Acht gelassen. Mit Sicherheit ist hier in erster Linie die Absicherung gegen Fremdzugriff (fremde Agenten) und DOS-Attacken gemeint. Farsight ist jedoch sicher unter dem parallelen Zugriff verschiedener Prozesse (*Thread Save*), was jedoch eigentlich eher unter die Rubrik Korrektheit fällt und hier nur erwähnt werden soll um Mißverständnissen vorzubeugen. Die Verwendung von Farsight sowie der Zugriff auf Farsight durch fremde Agenten sollte in Zukunft idealer Weise über die SeMoA-Policy [2] gesteuert werden.

Für das Filtern der eingehenden Kontakte steht eine Methode (*FarSightImpl.filter()*) bereit, die jedoch noch nicht vollständig implementiert ist.

Bisher ist das Entfernen von Kontakten aus FarSight nicht vorgesehen. Eine Implementierung dieser Funktionalität würde sicherlich die Flexibilität von Farsight erhöhen, dies sollte allerdings nur unter Verwendung der SeMoA-Policy geschehen.

Refactoring: Die Funktionalität eines Verzeichnisdienstes sollte vielleicht aus *FarSightImpl* in eine eigene Klasse *FarSightDirectory* ausgelagert werden. Das dient der besseren Verteilung der Verantwortlichkeiten der einzelnen Objekte (*Separation of Concerns*) und wird weitere Modifikationen erleichtern.

Eine weitere ist Idee für eine Discovery-Strategie ist die Verwendung von IP-Multicasting. Damit sollte sich ein vergleichbarer Effekt wie bei der Verwendung von Vicinity erzielen lassen.

8 Literatur

1. SeMoA Project. *SeMoA-API*. http://www.semoa.org/docs/api/index.html
2. V. Roth, U. Pinsdorf. *SeMoA Installation and Configuration Guide.* http://www.semoa.org/docs/install.pdf
3. Sun Microsystems. *JDK-API*. http://java.sun.com/j2se/1.4.1/docs/api/
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*, Addison-Wesley, 1994
5. P. Hyde. *Java Thread Programming*. Sams Publishing, 1999
6. A. Oram (Hrsg.). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.
7. M. Harchol-Balter, T. Leighton and D. Lewin. *Resource Discovery in Distributed Networks.* ACM, 1999.
8. C. Huitema. *Routing in the Internet*. Prentice Hall PTR, 1995.
9. S. Saroiu, P. K. Gummadi, S. D. Gribble, *A Measurement Study of Peer-to-Peer File Sharing Systems.* Tech. Rep. # UW-CSE-01-06-02, University of Washington.
10. V. Roth, U. Pinsdorf. *Mobile Agent Interoperability Patterns and Practise.* In Proceedings of Ninth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2002), Computer Graphics Edition, pages 238- 244, University of Lund, Lund, Sweden, April 2002.
11. Y. Rekhter, T. Li. *A Border Gateway Protocol 4 (BGP-4)*. RFC 1771, March 1995.
12. J. Moy. *OSPF Version 2*. RFC 2328, April 1998.
13. Freenet. http://freenet.sourceforge.net
14. SETI@home. http://setiathome.ssl.berkeley.edu
15. Napster. http://www.napster.com

16. Gnutella. http://www.gnutella.com

17. Z. Su, J. Postel. *The Domain Naming Convention for Internet User Applications.* RFC 819, August 1982.

18. I. Stoika, R. Morris, D. Karger, M. F. Kasshoek, H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications.* In Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California, August 2001.

19. A. Rowstron, P.Druschel. *Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems.* In international Conference on Distributed Systems Platforms (Middleware), November 2001.

20. M. Castro, P. Druschel, A.M. Kermarrec, A. Rowstron. *One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks.*

21. S. Deering. *Host extensions for IP multicasting.* RFC 1112, August 1989.

22. C. Shirky. *What is P2P ... And What Isn't ?.* http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html