Ludwig-Maximilians-Universität
Institut für Informatik
München

Technische Universität
Institut für Informatik
München

Lastverteilung und Ausfallsicherheit durch das Konzept des Virtuellen Servers

Fortgeschrittenenpraktikum/Systementwicklungsprojekt

Von
Benjamin Fingerle und Michael Krause

Aufgabensteller:
Prof. Dr. Heinz-Gerd Hegering

Betreuer:
Igor Radisic (LMU)
Sven Angerer (IconParc GmbH)

Inhaltsverzeichnis

Inhaltsverzeichnis 2

1 Einleitung 3

1.1 Motivation 3

1.2 Aufgabenstellung 3

2 Anforderungsanalyse 4

2.1 Szenarien 4

2.2 Anforderungskatalog 6

2.3 Der IPvise 6

3 Design 7

3.1 Status Quo 7

3.2 Architekturentwurf 11

4 Implementierung und Test 21

4.1 Details 21

4.2 Test 27

5 Zusammenfassung 30

6 Literatur- und Softwareverzeichnis 31

6.1 Literatur 31

6.2 Software 31

7 Abbildungsverzeichnis 32

1 Einleitung

1.1 Motivation

Zwei Schlagworte sind es, die dieses Projekt charakterisieren. Beide werden immer wieder in unterschiedlichstem Zusammenhang verwendet, und noch häufiger von den Marketingabteilungen der IT-Unternehmen aus demselben gerissen. Ausfallsicherheit ist das eine, Lastverteilung das andere.

Wie bei vielen Internet-Auftritten kann die Zahl der Zugriffe so groß werden, dass ein einzelner Rechner mit der Abarbeitung überfordert ist: Microsoft etwa setzt für seinen Hotmail-Service über 30000 Computer ein.

Einen einzelnen Rechner lediglich auszubauen, mit mehreren und schnellen Prozessoren zu versehen, kann bei wachsenden Zugriffszahlen allein aus Kostengründen nichts weiter als eine Not- oder Übergangslösung sein. Die Zeiten, in denen es billiger war, einen Rechner mit möglichst großer Leistung zur Verfügung zu stellen, statt mehrere, kleinere zu verwenden, ist dank der technischen Entwicklung und dem damit einher gehenden Preisverfall vorbei. Heute ist es weit teurer, die Kapazität eines einzelnen Rechner auszubauen, statt dieselbe gesteigerte Leistung durch eine höhere Zahl von Rechnern bereitzustellen, ganz davon abgesehen, dass jeder Rechner auch einmal an die Grenzen seiner Erweiterungsfähigkeit stößt.

Wirtschaftlicher und Zweckmäßiger ist also der Einsatz mehrerer Rechner, nur müssen diese auch koordiniert werden. Hier setzt dieses Projekt an. Im Folgenden der Wortlaut der Aufgabenstellung.

1.2 Aufgabenstellung

Lastverteilung und Ausfallsicherheit durch das Konzept des Virtuellen Servers:

Innerhalb des Projektes sei ein Modulset zu einem bestehenden Applikations-Serversystem zu entwickeln, das

- die Last nach definierbaren Strategien auf verschiedene Server verteilt,
- eine hohe Ausfallsicherheit garantiert,
- und dabei plattformübergreifend auf Windows-NT-, Solaris- und Linux-Rechnern lauffähig ist.

Dafür soll zunächst ein abstraktes Konzept erstellt und anschließend implementiert werden. Schließlich soll der entstandene Virtuelle Server unter Realbedingungen getestet und Konzept wie Implementierung dokumentiert werden.

Ziel ist es, sinnvolle Funktionseinheiten in Programmmodule zusammenzufassen, die je nach Anwendungszweck unabhängig voneinander verwendet und kombiniert werden können.

2 Anforderungsanalyse

2.1 Szenarien

2.1.1 Ein Virtueller Server

Aus Sicht des Clients, der auf die Web-Applikation zugreift, muss die gewachsene Zahl der Rechner transparent bleiben, er darf weiterhin nur eine Anlaufstelle kennen, sprich eine einzige Internetadresse. Von dort müssen die Anfragen auf die zur Verfügung stehenden Rechner verteilt werden, am besten abhängig von deren bisheriger Auslastung. Dies erledigt die in diesem Projekt erarbeitete Lastverteilung. Auf Nutzerseite bleibt weiterhin die Sicht auf einen, - wenn auch nur noch – virtuellen Server (Abb. 1) bestehen.

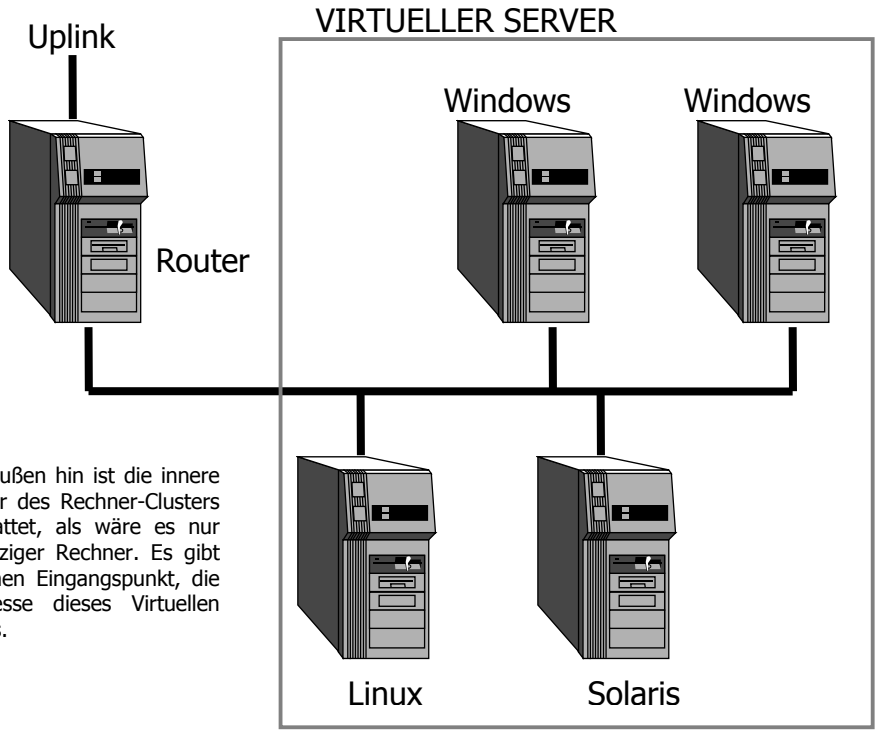


Abb. 1: Konzept des virtuellen Servers

Je mehr Zugriffe auf eine Website erfolgen, desto schmerzlicher sind deren Ausfälle – ein System, dass sich für besonders große Zugriffszahlen rüstet, muss sich auch dazu Gedanken machen. Ausschließen wird man eine Störung oder den Ausfall eines Rechners nie können. Aber diese Situation muss schnellstmöglich erkannt werden, um zu verhindern, dass diesem Rechner weitere Anfragen zur Abarbeitung zugeteilt werden, die dann verloren gingen. Geradezu katastrophal wäre ein Ausfall des Rechners, der die Anfragen aus dem Web entgegennimmt und verteilt, denn dann stünde der gesamte Rechnercluster außer Betrieb. Auch hier müssen im Rahmen dieses Projekts Strategien erarbeitet werden, die verhindern, dass es innerhalb des Rechnerclusters einen so genannten „single point of failure“ geben kann und den Rechnerverbund überhaupt zügig auf etwaige Ausfälle

in den eigenen Reihen reagieren lassen. Dies ist gemeint, wenn von der Ausfallsicherheit des Virtuellen Servers gesprochen wird.

Wir gehen dabei von folgender Beispiel-Konstellation aus: Wir haben vier Rechner - gleichbestückte x86-PCs – von denen zwei mit Windows2000, einer mit Suse Linux 7.3 und ein weiterer mit Solaris als Betriebssystem ausgestattet ist. Dabei ist gleich vorwegzunehmen, dass Solaris als einziges Betriebssystem so starke Probleme bereit hat, dass es nicht gelang, das System auch dafür auszurüsten, und sich das Testsystem deshalb auf Linux und Windows2000 beschränken musste.

Vom Router-/Firewallsystem, erwarten wir eine höchstmögliche Funktionsgarantie, es stellt in unserer Umgebung des Virtuellen Servers die einzige Schnittstelle des Server-Verbundes zum Internet dar. Etwaige Möglichkeiten, ein solches Router-/Firewallsystem beispielsweise durch Parallelisierung oder andere Maßnahmen verfügbarer zu gestalten, werden wir im Rahmen dieses Projektes nicht behandeln. Sprich: Die Ausfallsicherheit, die dieses Projekt liefern soll, setzt erst nach der Router/Firewall-Instanz an.

Des weiteren gehen wir davon aus, dass die Rechner über ein Broadcast-Medium wie Ethernet (IEEE 802.3) kommunizieren.

2.1.2 Der WebMediator

Der hier vorgestellte Virtuelle Server sollte am Beispiel des WebMediators der Münchner IconParc GmbH entwickelt und getestet werden, aber prinzipiell genauso gut mit anderen Anwendungen verbunden werden können.

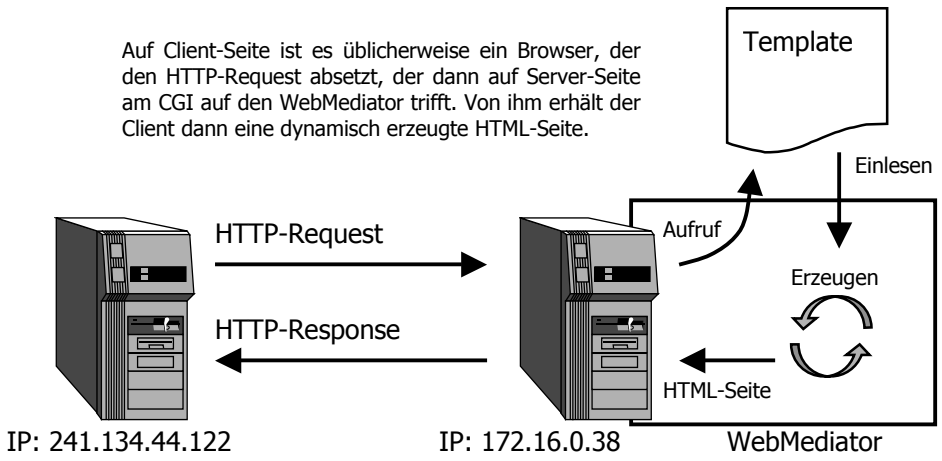


Abb. 2: Arbeitsweise des WebMediators nach [Brüc02]

Der WebMediator ist ein Java-basierter Web-Applikationsserver, der dynamische HTML-Seiten erzeugen kann (Abb. 2). Mit ihm lassen sich zum Beispiel E-Shops oder Webbasierte Redaktionssysteme verwirklichen.

2.2 Anforderungskatalog

Zusammenfassend seien hier noch einmal die Anforderungen an die zu entwickelnde Software zusammengestellt, die zum größten Teil ja bereits in der Aufgabenstellung vorgegeben waren.

- Es ist eine Strategie notwendig, um zentral Informationen über die momentane Last der Rechner zu bekommen, heuristisch oder geschätzt.

- Auf Basis dieser Werte ist eine gleichmäßige Verteilung der Last zu erreichen.

- Skalierbarkeit: Da bei hochfrequentierten Internet-Applikationen ein Wachsen des Traffics zu erwarten ist, und so auch ein Ausbau des Verbunds einkalkuliert werden muss, ist es zweckmäßig, diesen Verbund eigenstabil zu gestalten, damit das System auch bei Hinzufügen weiterer Rechner verfügbar bleibt.

- Redundanz: Um beim Ausfall eines Rechners weiterhin für den vollen Funktionsumfang garantieren zu können, muss ein anderer Rechner in der Lage sein, dessen Aufgaben zu übernehmen. Somit sollte jeder Teil der Software mindestens auf mehreren, wenn nicht gar auf allen Rechnern verfügbar sein – und etwa rollenabhängig zum Einsatz kommen.

- Es sind Strategien notwendig, um den Ausfall einer Komponente den anderen Rechnern anzuzeigen und sodann den Verbund neu zu organisieren.

- Ebenso ist eine Strategie notwendig, den Router wissen zu lassen, an welchen Rechner er die Anfragen zu leiten hat, wenn sie an die IP-Adresse des Verbundes gesandt wurden. Auch muss dafür gesorgt werden, dass diese Adresse wechseln kann, und dieser Wechsel dem Router entsprechend angezeigt wird.

- Schließlich sind Schnittstellenmethoden zu implementieren, die die Funktionalität der eigentlichen Web-Applikation zur Verfügung stellen, und zudem Bedienung und Wartung ermöglichen.

2.3 Der IPvise

Projektname für die zu entwickelnde Software ist „IPvise“ als Akronym für „IconParc Virtual Server“.

3 Design

3.1 Status Quo

3.1.1 Programmiersprache

Aus Gründen der Plattformunabhängigkeit und der Anbindung an den Beispielserver ist Java die Programmiersprache der Wahl. In Verbindung damit kommt auch C++ gemeinsam mit dem Java Native Interface zum Einsatz.

3.1.2 Lastverteilung

3.1.2.1 Vorgehensweisen

Rechenarbeit auf mehrere Rechner, eigentlich mehrere Rechnerkerne zu verteilen, verkürzt die Antwortzeit und erhöht so die Zahl der Prozesse, die in einem festen Zeitraum abgearbeitet werden können.

Das Verfahren ist nicht trivial: Computerprozesse bedürfen der Synchronisierung, die teils sehr aufwändig werden kann. Und nicht nur das; Prozesse, die abgearbeitet sind, geben Speicherplatz für nachfolgende frei.

Allgemein gesprochen hängt nach [Ludw96] die von einem Mehrprozessorsystem erreichbare Leistung in entscheidendem Maße ab von der Abbildung der lasterzeugenden Objekte (Prozesse und ähnliche programmiersprachliche Konstrukte) auf die leistungserzeugenden Objekte (Rechnerkerne aller Art). Dabei sieht er drei „Zeitpunkte“ (besser: Phasen), während der der Behandlung der Objekte besondere Bedeutung zukommt:

- Vor dem Start eines Programms sind die potenziell nebenläufig ausführbaren Teile dieses Programms zu identifizieren, ein Vorgang, den man **Partitionieren** nennt.

- Ebenfalls vor Ausführung des Programms erfolgt die **Rechnerknotenzuteilung** mit dem Ziel, aus Benutzersicht die Verweildauer eines Objektes auf einem Rechnerknoten zu minimieren. Pro Objekttyp wird nach [Ludw96] meist mindestens ein Objekt erzeugt, das einem konkreten Rechnerknoten zugeteilt wird. Dabei ist ein Kompromiss zwischen den sich widersprechenden Zielen der Minimierung, der Interprozess-kommunikation auf der einen und der Optimierung der gleichmäßigen Knotenauslastung auf der anderen Seite zu finden.

- Die **Lastverwaltung** entspricht einer Rechnerknotenzuteilung zur Laufzeit, wobei noch einmal unterscheiden wird zwischen der **Prozessplatzierung**, also einer Zuteilung von neu entstehenden Objekten und der **Prozessverschiebung**, bei der Objekte, die bereits im System Last erzeugen, auf andere Knoten bewegt werden. Letzteres ist, wie man sich leicht vorstellen kann, technisch aufwändiger.

Das prinzipielle Vorgehen entspricht nach [Ludw96] den typischen Abläufen eines Regelkreises (Abb.3): ein Messfühler erfasst den aktuellen Zustand (die Last) des Systems, eine Reglereinheit ermittelt die Abweichung vom Sollzustand (der optimalen Maschinennutzung) und versucht, über ein Stellglied diesen wieder herzustellen.

Dabei gibt es verschiedene Probleme zu beachten, darunter:

- die Optimierung der Reaktionsgeschwindigkeit des Regelkreises,
- das Vermeiden von Überreaktionszeiten beim Ausregeln und
- das Verhindern oszillierenden Verhaltens.

Weiterhin erzeugt auch der Regelkreis selbst Last und muss in die Bewertung des Systems miteinbezogen werden.

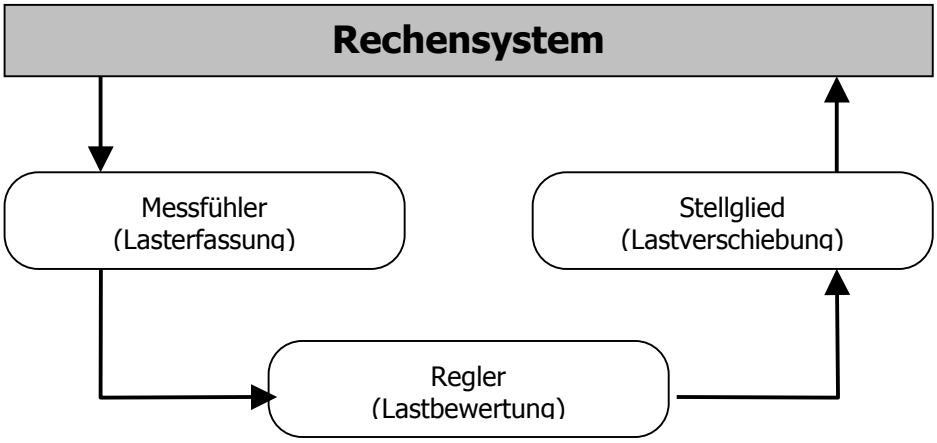


Abb. 3: Phasen der Lastverteilung nach [Ludw96]

3.1.2.2 Partitionierung der Last

Ein lasterzeugendes Objekt ist in diesem Zusammenhang die Abarbeitung einer http-Anfrage, für die der WebMediator ein Skript parst, interpretiert, im Zuge dessen eventuell Datenbank-Operationen ausführt und schließlich als Ergebnis eine dynamisch erzeugte html-Seite zurückgibt. Die Rechenzeit lässt sich prinzipiell nicht vorhersagen. Da aber die Entwickler der Skripte die Antwortzeiten so kurz wie möglich halten wollen, können wir vereinfachend davon ausgehen, dass sie einen gewissen Schwellwert bei der Rechenzeit nicht überschreiten und diese also zwischen zwei verschiedenen http-Anfragen nicht signifikant abweicht.

Als lasterzeugende Objekte kleinere Einheiten zu wählen als http-Anfragen ist nicht sinnvoll, da im Vergleich dazu der Overhead der Lastverteilung viel zu groß wäre. Selbst jeden neu eintreffenden einzelnen Request zu entscheiden, wo er abgearbeitet werden soll, bringt mehr Verwaltungsaufwand als Nutzen.

Der Blick auf die Arbeitsweise des WebMediators lässt noch größere Einheiten ratsam erscheinen: Die jeweilige Benutzersession. Beim ersten

Request an das Framework des WebMediators erhält jeder Benutzer einen Sessionkey, anhand dessen seine Anfragen als zu seiner Sitzung gehörig erkannt wird. Der Benutzer kann sich in dieser Sitzung identifizieren, per Login und Passwort authentifizieren, Rechte erhalten, in seiner personalisierten Umgebung Informationen abrufen oder Bestellungen tätigen. Erfolgt eine bestimme Zeit lang kein Request mit diesem Sessionkey, erklärt das WebMediator-Framework die Sitzung für beendet. Da die Informationen über den Status einer Sitzung im Speicher des Rechners vorgehalten werden, wäre es unwirtschaftlich, http-Anfragen, die zur selben Session gehören auf unterschiedlichen Rechnern abzuarbeiten. Wir erklären also eine Session zu einem lasterzeugenden Objekt und verfolgen demnach das Ziel, die Sessions ergonomisch auf die zur Verfügung stehenden Rechner zu verteilen.

3.1.2.3 Rechnerknotenzuteilung

Vor Laufzeit des IPvise sind über Zahl und Zeitpunkt der http-Anfragen bestenfalls Schätzungen möglich. Die Rechnerknotenzuteilung zu diesem Zeitpunkt beschränkt sich daher auf die Planung der Abarbeitung des IPvise selbst, der ja – wie im einleitenden Abschnitt erwähnt – ebenfalls Last erzeugt. Um den Verbund von Rechnern, der den virtuellen Server bildet, eigenstabil zu halten, und auch die Übertragung der Dispatch-Aufgabe zu ermöglichen, läuft der IPvise auf jedem einzelnen Priest eigenständig, wie an späterer Stelle, an der die Ausfallsicherheit erläutert wird, noch dargelegt wird.

Es ist Wunsch des Kunden, dass der Pope sich auf das Dispatchen beschränkt, um einen möglichst großen Leistungs-Puffer für die Lastverteilung selbst zu haben und nicht zum Flaschenhals zu degenerieren. Nur für den pathologischen Fall, dass die Priests nicht mehr erreichbar wären, übernimmt der Pope selbst die eingehenden Requests, um zumindest einen Notbetrieb aufrecht zu erhalten.

3.1.2.4 Lastverwaltung

Abgesehen vom eben behandelten IPvise selbst, gibt es nach unserer Definition die Sessions als Menge von http-Anfragen als Typ eines lasterzeugenden Objekts, die allerdings unvorhersehbar erst zur Laufzeit eintreffen. Bereits aus der Partitionierung ergibt sich, dass der IPvise im Sinne von [Ludw96] bezogen auf die Sessions eine reine Prozessplatzierung betreibt, weil eine Prozessverschiebung im Verhältnis zum Nutzen zu aufwändig wäre. Im oben erwähnten Regelkreis (Abb 3) der Lastverwaltung ist mit der Umleitung der initialen http-Anfrage die Realisierung des Stellwerks definiert, bleiben noch Messfühler und Regler.

Die Ermittlung der Last zerfällt in zwei Teilprobleme. Zum einen die Definition und Ermittlung der Last eines Rechners, zum anderen die Bekanntgabe derselben an den jeweiligen Pope. Mit dem ersten Problem beschäftigen wir uns nicht weiter, weil es zweckmäßiger bei der Software angesiedelt ist, deren Last zu verteilen ist. Dort finden sich die Informationen, um zu entscheiden, welche Ressourcen diese benötigt und belegt, und dort kann auch die freie Kapazität ermittelt werden.

3.1.2.5 Algorithmus der Lastverteilung

Im Kern der erläuterten Lastverteilungsstrategie geht es um die Frage, welcher Rechner die Session der neu eingetroffenen http-Anfrage übernimmt. Das ist die Entscheidung, die der Algorithmus lösen muss.

Dabei spielen eine Reihe von Faktoren mit, die sich nicht vorhersagen lassen, sondern erst zur Laufzeit ermittelt werden können. Das sind zum einen Zahl und Zeitpunkt der eintreffenden http-Anfragen (gemeint sind hier die initialen Anfragen, die den Pope erreichen, nicht die innerhalb einer Session folgenden http-Anfragen, die direkt an den jeweils zuständigen Priest geschickt werden), genauso wenig ist a priori bekannt, wie viele Rechner zur Verfügung stehen, wie leistungsfähig sie sind, und wie viel dieser Leistung für den IPvise zur Verfügung stehen.

Eine mögliche Strategie wäre das Round-Robin-Verfahren, damit würden alle bekannten Priests reihum eine Session zugeteilt bekommen, danach müsste der Pope neu entscheiden, welche Priests ihm zur Verfügung stehen – es könnte einer ausgefallen sein, oder einer neu hinzugekommen – und dann wieder von vorn mit der Zuteilung beginnen. Diese Strategie ist sicher die einfachste. Sie vernachlässigt allerdings, dass zum einen die Rechner von unterschiedlicher Kapazität sein, zum anderen einige Session weit größere Rechenlast erzeugen können als andere. Die Auslastung wäre in beiden Fällen alles andere als optimal.

Ins andere Extrem könnte folgender Algorithmus degenerieren: Man stelle sich vor, bei Eintreffen einer http-Anfrage verlange der Pope erst einmal von allen Priests Mitteilung über ihre momentane Kapazität und teile die http-Anfrage dann demjenigen mit der größten Leerlast zu. Diese Strategie würde zwar ausgezeichnet für eine gleichmäßige Lastverteilung sorgen, aber zu welchem Preis: Da die http-Anfragen in hoher Frequenz eintreffen können, würde die notwendige Kommunikation das Netz kräftig belasten und die Antwortzeiten in nicht mehr tolerierbarem Maße verlängern.

3.1.3 Ausfallsicherheit

Zunächst einmal müssen wir klären, was wir unter einem ausfallsicheren System verstehen, über welche Komponenten dieses Systems sich die Ausfallsicherheit erstrecken soll, welche Art von Ausfällen es zu überbrücken gilt:

Wir gehen von folgender Konstellation aus: Wir haben mehrere Server-Systeme sowie einen Router/Firewall in Hardware oder (wie in unserer Testumgebung) als umfassend getestete Software-Variante.

Vom Router/Firewallsystem erwarten wir eine höchstmögliche Funktionsgarantie, es stellt in unserer Umgebung des Virtuellen Servers die einzige Verbindung des Server-Verbundes zum Internet dar. Etwaige Möglichkeiten, ein solches Router/Firewallsystem beispielsweise durch Parallelisierung oder andere Maßnahmen verfügbarer zu gestalten, werden wir im Rahmen dieses Projektes nicht behandeln.

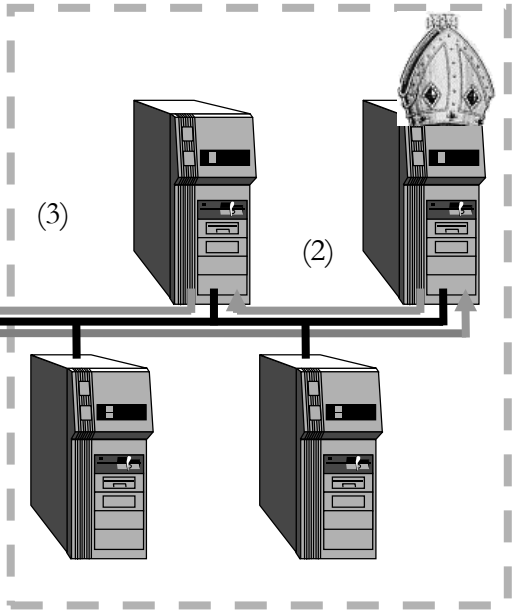
Uns bedeutet die Ausfallsicherheit des virtuellen Servers, eine höchstmögliche Verfügbarkeit der gebotenen Anwendung zu gewährleisten, indem wir die Funktionalität des Verbundes vor den Ausfällen einzelner Elemente, also Serversysteme, so weit wie möglich schützen. Dabei steht in allen Belangen das Prinzip der Mehrheitsentscheidung im Vordergrund: Keine Entscheidung - sei es, den Dispatcher zu bestimmen, sei es, den aktuellen Dispatcher als nicht einwandfrei funktionierend abzusetzen, oder auch, einen neubestimmten Dispatcher dem Router/Filesystem mitzuteilen - all diese die Funktionalität des Virtuellen Servers beeinflussenden Entscheidungen müssen von der Mehrheit getroffen werden. Ein Einzelner darf nicht eigenmächtig handeln, da wir von einzelnen Serversystemen jeden nur denkbaren pathologischen Zustand erwarten. Dabei kann es sich im einfachsten Fall um einen kompletten Ausfall handeln, aber auch der Verlust einzelner Fähigkeiten, beispielsweise durch Verklemmung einzelner Threads, oder auch scheinbar korrektes, im Detail jedoch auf falschen Annahmen beruhendes Verhalten eines einzelnen Serversystems gilt es aufzudecken und gegebenenfalls zu beheben, notfalls aber auch diesen komplett vom Verbund auszuschließen. Ein Fehlverhalten der Form, dass eines der Serversysteme eine Mehrheitsbestimmung fälschlicherweise als gegeben ansieht und daraus die Legitimation zur Vollstreckung des Mehrheitsbeschlusses ableitet, muss in jedem Falle von der Gruppe als solches erkannt und adäquat behandelt werden. Mit der Architektur des IPvise ist es zwar nicht möglich, jeden Fall funktionsstörenden Verhaltens gänzlich zu verhindern, wohl aber, die Funktion des Gesamtsystems zumindest meistzeitlich wiederherzustellen.

3.2 Architekturentwurf

3.2.1 Lastverteilung

(1) Initialer http-request an die Adresse des Virtuellen Servers

(2) Pope verteilt die Anfrage



(3) Ausgewählter Priest übernimmt Anfrage und erzeugt Session

(4) http-Antwort von Priest, alle weiteren Anfragen in dieser Session an ihn

Architekturentwurf 1: Ein ausgezeichneter Rechner (Pope) übernimmt allein die Verteilung. Die anderen Rechner (Priests) erhalten die Sessions.

3.2.1.1 Vorgehensweise

Realisiert wird die Lastverteilung wie folgt: Die erste http-Anfrage des Benutzers geht an die IP-Adresse des Dispatchers, dieser antwortet mit einer Umleitungsanweisung an den Browser zu dem Rechner, den der Dispatcher mittels seiner Lastveteilungsstrategie als derzeit günstigsten ausgewählt hat. Alle weiteren http-Anfragen zu dieser Session (wie Links oder Form-Submits) beziehen sich relativ auf diese Adresse.

In der IPvise-Terminologie sind alle Rechner des virtuellen Servers „Priests" und der eine, der gerade mit der Verteilaufgabe betraut ist, der „Pope".

Aufgabe des IPvise ist es in diesem Zusammenhang, dem WebMediator eine Methode bereitzustellen, die dem Framework bei Eintreffen einer http-Anfrage an den Pope die IP-Adresse des Priests liefert, der alle Anfragen aus dieser neuen Session erhalten soll (PENDING: vgl. Abschnitt „Definition der Schnittstelle").

Damit ist die Partitionierung im Sinne von [Ludw96] hinreichend definiert.

3.2.1.2 Lastverwaltung

Der IPvise erwartet vom WebMediator oder der Software, zu deren Verteilung er eingesetzt wird, eine Methode, die ihm einen Wert für die momentan freie Kapazität übergibt, den er dann für seinen weiteren Lastverteilungsalgorithmus benutzen kann. Einzige Festlegung: Der Wert ist eine positive Ganzzahl, und desto größer, je mehr freie Kapazität zur Verfügung steht. Es versteht sich, dass auf allen Rechnern der gleiche Algorithmus zur Ermittlung der Kapazität angewandt werden muss, um diese Werte überhaupt vergleichen zu können.

Die Bekanntgabe an den Pope ist ein Push-Mechanismus: Jeder Priest sendet in regelmäßigen Abständen einen mit seiner IP-Adresse gekennzeichneten Broadcast mit seiner derzeitigen freien Kapazität ins Netz. Darüberhinaus stellt er dem WebMediator eine Methode zur Vefügung, selbst einen solchen Broadcast auszulösen, falls er eine signifikante Laständerung erkannt hat, die er sofort und nicht erst mit der nächsten turnusgemäßen Lastmitteilung dem Pope bekannt geben will. Die Gesamtheit der eingehenden Lastmitteilungen gibt dem Pope nicht nur ein Bild der für ihn verfügbaren Kapazität, sie teilt ihm auch mit, welche Priests sich derzeit in dem Rechnerverbund des virtuellen Servers befinden. Empfängt er von einem bestimmten Priest keine Lastmitteilung mehr, geht er davon aus, dass dieser nicht mehr verfügbar ist und berücksichtigt ihn nicht weiter.

Der Vollständigkeit halber sei an dieser Stelle auch erwähnt, dass in diesen Lastmitteilungen auch Daten über die IPvise-Konfiguration des Absender-Rechners stehen, um das Debuggen zu vereinfachen. Außerdem ist geplant, diesen Mitteilungen auch Informationen über Gültigkeitsbereiche der Sessionkeys mitzugeben, so dass bereits der Pope der initialen http-Anfrage einen Sessionkey zuweisen kann, und dies nicht erst auf dem Priest

geschieht, an den die Anfrage umgeleitet wird. Doch dies soll nicht mehr im Rahmen dieses Projekts geschehen.

3.2.1.3 Algorithmus zur Lastverwaltung im IPvise

Der Mittelweg zwischen Round Robin Verfahren und permanentem Zustandspolling durch den Priest, den der IPvise geht, vereint Elemente von beidem und lehnt sich an das Priority-Scheduling der Zuteilung der Prozesse an den Rechnerkern auf Betriebssystem-Ebene an. Um die wichtigsten Elemente zu nennen: Die Priests werden anhand der regelmäßig von ihnen übermittelten Angaben über ihre freie Kapazität dynamisch in Klassen eingeteilt (Abb. 4, „Lastmitteilung“) und die eintreffenden Requests werden im Round-Robin-Verfahren an die Priests der jeweils höchsten nichtleeren Klasse verteilt (Abb. 4, „Dispatch“).

Dabei ist zu beachten, dass die Lastmitteilungen der Priests nicht synchronisiert, also die Informationen über ihre freie Kapazität unterschiedlich aktuell sind. Der Pope muss also wissen, wie alt die Kapazitätsinformation eines Priests ist, um ihn entweder in eine tiefere Prioritätsklasse (in unserem Modell heißen diese „Pews“) einzuteilen, sobald er entsprechend viele http-Anfragen zugewiesen bekommen hat, dass davon auszugehen ist, dass seine freie Kapazität inzwischen signifikant gesunken sein dürfte. Oder der Pope muss Priests, von denen keine neue Lastmitteilung eintrifft ganz aus seiner Lastverwaltung ausschließen, da er davon ausgehen muss, dass diese entweder nicht mehr am Netz hängen oder aus anderen Gründen nicht mehr funktionstüchtig sind (Abb. 4 „Timeout“).

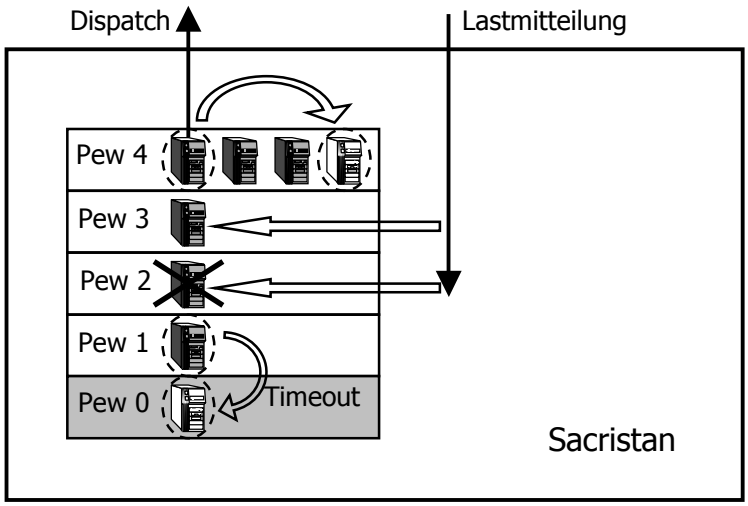


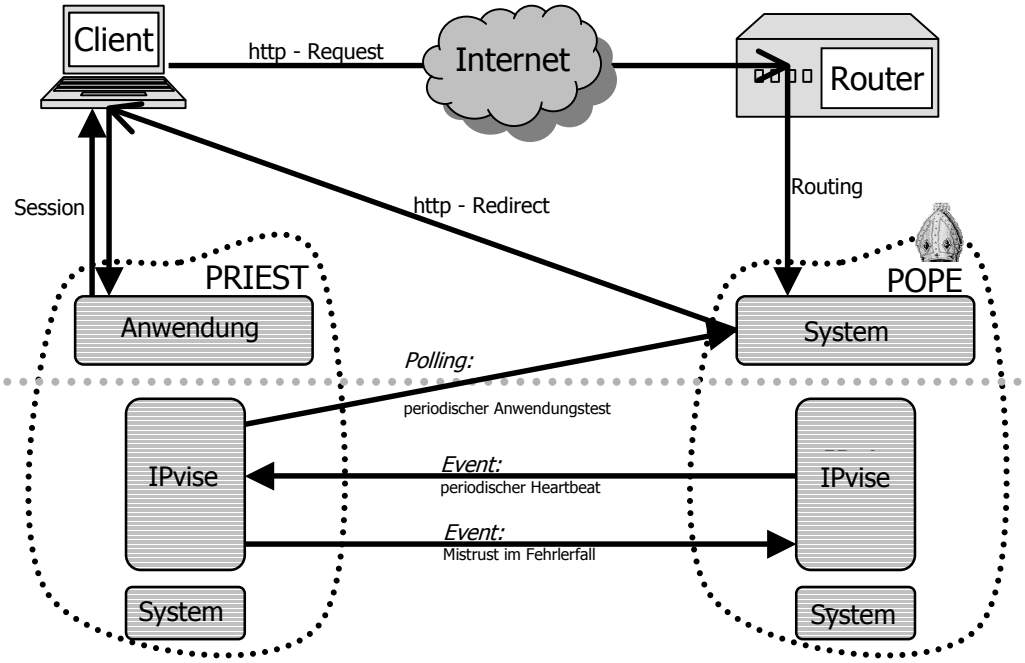
Abb. 4: Dynamische Multilevel Feedback Queues

Auch ist weiter zu beachten, dass zwar die Zahl, keineswegs aber die Definition der Prioritätsklassen statisch sein darf. Schließlich weiß das System nicht a priori, welche Werte für die Beschreibung der freien Kapazität auftreten können. Zur Erinnerung: In der Schnittstellendefinition zum WebMediator (oder der entsprechenden Applikation mit der der IPvise verbunden wird) ist lediglich verlangt, dass die Kapazität durch eine positive Ganzzahl beschrieben wird, die desto höher ist, je größer die freie Kapazität

ist. Wie soll also entschieden werden, welcher Kapazitätswert auf welche Pew abgebildet wird? Es kann nicht zweckmäßig sein, dass ein Priest mit der Kapazität 50 der höchsten Pew zugewiesen wird, wenn dort auch ein Priest der Kapazität 10000 eingeordnet wurde. Das Problem wird wieder zur Laufzeit gelöst. Der Pope nimmt die höchste eingegangene Lastmitteilung als Maßstab für die Obergrenze der Pews.

3.2.2 Ausfallsicherheit

Um die geforderte Ausfallsicherheit zu erreichen, wird es im Rahmen des IPvise dem Verbund ermöglicht, die Funktionalität des aktuellen Dispatchers zu kontrollieren, einen vermeintlich ausgefallenen Dispatcher abzusetzen und einen neuen Dispatcher als solchen nach vorangegangener Wahl beim Router/Firewallsystem anzumelden. Im folgenden wird beschrieben, auf welchen Daten eine solche Wahlentscheidung basiert und wie sich die Rolle eines einzelnen Servers in solchen Mehrheitsbeschlüssen und -handlungen gestaltet.



Architekturentwurf 2: Die Zustandsüberwachung des virtuellen Servers ermöglicht ein Event-/Polling-basiertes Mischmodell.

Verschiedene Moduln erwirken zusammen das gewünschte Verhalten:

3.2.2.1 Das Heartbeatmodul

Um dem Rest des Verbundes mitzuteilen, dass der aktuelle Dispatcher sich weiterhin in der Lage sieht, die gestellten Anforderungen zu erfüllen, wird das bekannte Heartbeat-System verwandt: Der Dispatcher sendet in vorgeschriebenen Intervallen periodisch eine Meldung. Diese Meldung beinhaltet seine Identität (in Form der eindeutigen IP-Adresse) gekoppelt an eine Prioritätskennzahl, kodiert zu einer ebenfalls eindeutigen Ganzzahl, auf der durch den gewöhnlichen '>='-Operator eine Ordnung definiert ist, die einen mit höherer Priorität versehenen Server dem mit niedrigerer Priorität

voranstellt. Sollte nun der Dispatcher gänzlich ausfallen, so ist es dem Verbund möglich, das Ausbleiben der Heartbeat-Meldungen über mehrere Takte hinweg (wir nennen fortan die festgelegt Länge des Intervalls zwischen zwei Heartbeat-Meldungen einen Takt) zu bemerken und die Neuwahl sowie die Anmeldung beim Router/Firewallsystem zu vollführen.

Doch wie soll der mögliche pathologische Fall behandelt werden, bei dem der Dispatcher in der Art ausfällt, sich selbst für funktionsfähig zu halten, obwohl er seine Aufgaben nicht zu erfüllen vermag, wie soll es bemerkt, wie die weiterhin gesendeten Heartbeat-Meldungen behandelt werden? Für die Identifikation dieses Zustands ist das weiter unten beschriebene Applicationcheck-Modul verantwortlich, weshalb wir an dieser Stelle davon ausgehen, dass der Verbund die Fehlfunktion zu bemerken in der Lage ist.

Um die „falschen" Heartbeat-Meldungen zu „entlarven", wird folgender Mechanismus verwendet: In jede Heartbeat-Meldung wird zusätzlich eine Kennung kodiert, bei der es sich um eine Ganzzahl handelt, die zu Beginn des Betriebs des virtuellen Servers mit null initialisiert wird. Stellt der Verbund mehrheitlich fest, dass der aktuelle Dispatcher nicht ordnungsgemäß arbeitet, so setzt ein jedes Element des Verbundes - also auch diejenigen, die die Fehlfunktion nicht selbständig erkannten - die eigene akzeptierte Kennung herauf, was bewirkt, dass alle Hearbeat-Meldungen mit einer enthaltenen Kennung kleiner als der eigenen akzeptierten Kennung schlicht ignoriert werden. Dies führt wiederum dazu, dass der Verbund keine (gültigen) Heartbeat-Meldungen mehr erhält und in Folge dessen einen neuen Dispatcher bestimmen wird.

Im Detail gestaltet sich diese „demokratische" Wahl wie folgt: Wurde durch den Verbund festgestellt, dass der Dispatcher defekt ist (wie dies geschieht, wird in der Beschreibung des Applicationcheck-Moduls erklärt), setzt jeder einzelne Server des Verbundes als Reaktion auf das Ausbleiben der Heartbeat-Meldungen seinen eigenen Status auf DISPATCHER und beginnt, selbst Heartbeat-Meldungen zu versenden. Diese sind also mit seiner eindeutigen Kennzahl, bestehend aus Priorität und IP-Adresse, sowie mit seiner aktuell akzeptierten Kennung versehen. Diese Verhaltensweise für sich führte natürlich zu einem abrupten Versand unzähliger Heartbeat-Meldungen und keiner eindeutigen Bestimmbarkeit des Dispatchers, weshalb ein Server den Versand von Heartbeat-Meldungen wieder einstellt, sobald er eine fremde Heartbeat-Meldung erhalten hat, die in erster Linie eine gleich hohe Kennung wie die eigene akzeptierte Kennung enthält UND von einem Server mit höherer Kennzahl, also höherer Priorität stammt. Enthält sie sogar eine höhere als die eigene akzeptierte Kennung, so beendet er den Versand von Heartbeat-Meldungen, ohne auf die Kennzahl zu schauen und übernimmt zusätzlich die neue akzeptierte Kennung. So bleibt am Ende dieses Wahlvorgangs ein eindeutig bestimmter Dispatcher übrig, den es dem Router/Firewallsystem vorzustellen gilt. Wie das geschieht, wird in der Beschreibung des Arp-Spoofing-Moduls beschrieben.

Erwähnenswert bleibt, dass der vermeintlich defekte Dispatcher die Chance hat, an diesem Wahlverfahren teilzunehmen: Erhält er die (eigenen) Heartbeat-Meldungen nicht mehr, so inkrementiert er ebenfalls die eigene akzeptierte Kennung, stellt gegebenenfalls fest, dass die Hearbeat-Meldungen ausbleiben und wird dann wie seine Mitstreiter versuchen, neue

Heartbeat-Meldungen zu versenden und so an der Wahl teilzunehmen. So schließen wir den defekten Dispatcher nicht vollkommen vom Betrieb des Verbundes aus (was bei vielen Servern durchaus zu verkraften wäre, in unserem Testaufbau jedoch schon zu großen Performanceeinbußen führte), auch, weil es nicht allzu unwahrscheinlich scheint, dass sich lediglich der Heartbeat-Meldung-Versand verklemmen und damit der betroffene Server nicht wieder zum Dispatcher werden könnte, wohl aber weiterhin als Teil des Verbundes, die Aufgaben eines Applikations-Servers leistend, zu agieren in der Lage wäre. Unserem Testaufbau bliebe so trotz dieses teilweisen Ausfalls seine volle Berechnungskraft erhalten.

Der Mechanismus, die Neuwahl des Dispatchers einzuleiten, sobald über einen längeren (konfigurierbaren) Zeitraum hinweg keine Hearbeat-Meldungen empfangen worden sind, greift selbstverständlich auch in dem Fall, dass der aktuelle Dispatcher auf natürliche Weise den Versand eben dieser beendet hat, sei es, weil er ausgeschaltet wurde, sei es, weil Windows seinen Dienst einstellte.

3.2.2.2 Das Applicationcheck-Modul

Aufgabe dieses Moduls ist es, wie oben bereits erwähnt, folgendes Fehlverhalten des Dispatchers aufzudecken:

Der Dispatcher bemerkt seine Fehlfunktion – zum Beispiel in Form von Zuweisungen falscher Server - nicht und weist weiterhin das Heartbeat-Modul an, regelmäßig Hearbeat-Meldungen zu versenden.

Um es den Servern zu ermöglichen, dieses fehlerhafte Verhalten des Dispatchers zu erkennen, wird auf jedem Teilnehmer des virtuellen Servers ein Applicationcheck-Modul ausgeführt. Hierbei handelt es sich um einen zusätzlichen Thread, der in einstellbaren Intervallen die vom Applikations-Server bereitzustellende Funktion isApplicationAlive() auf ihren booleschen Rückgabewert hin prüft. Ein negatives Ergebnis erwirkt das Versenden einer sogenannten Mistrust-Message per Broadcast an alle teilnehmenden Server. Eine Mistrust-Message beinhaltet die Kennzahl des sendenden Servers und wird von jedem Teilnehmer in einer Tabelle temporär gespeichert. Ihre Gültigkeit ist deshalb zeitlich beschränkt, weil einzelne Fehler in der Praxis durchaus vorkommen können und nicht eine sofortige Abwahl des Dispatchers erwirken dürfen. Zu diesem Zweck und nach dem oben erläuterten Prinzip der Mehrheitsentscheidung führt eine Mistrust-Message erst dann zu einer Neuwahl, wenn in den letzten k Takten mindesten von einem definierbaren Teil aller Server eine Mistrust-Message gesendet worden ist. Zudem veranlasst eine Mistrust-Message – auf Wunsch des Kunden – die sie empfangenden Server an, sofort einen eigenen Applicationcheck durchzuführen. Dieses Verhalten führt natürlich zu einem schlagartigen Anstieg der Dispatcher-Last, abhängig von der Anzahl der Teilnehmer. Sollte der erste negative Applicationcheck auf kurzzeitige Überlastung zurückzuführen sein, wäre es durchaus möglich, dass der Dispatcher sich erholte und fortan ohne Komplikationen seine Aufgabe fortsetzte, würde nicht durch die Synchronisation der Applicationchecks ein künstlicher „Peak“ der Last erzeugt.

Wie wird aufgrund von Mistrust-Messages eine Neuwahl des Dispatchers eingeleitet?

Analog zum Vorgehen des Heartbeat-Moduls bei ausbleibenden oder fehlerhaften Heartbeat-Messages, inkrementiert jeder Teilnehmer des Virtuellen Servers seine akzeptierte Kennung, sobald er von der Mehrheit in den letzten Takten eine Mistrust-Message erhalten hat. Diese Inkrementierung führt dazu, dass die eingehenden Heartbeat-Messages des aktuellen Dispatchers als fehlerhaft erkannt werden und in Folge nicht genügend gültige erhalten werden. Die Teilnehmer inkrementieren ihre akzeptierte Kennung abermals und beginnen selbst Heartbeats mitsamt neuer akzeptierter Kennung zu versenden, was dann auch von Servern bemerkt werden kann, die vom vorherigen Verfahren noch nichts mitbekommen haben, z.B. weil sie erst während des letzten Taktes dem Virtuellen Server zugefügt worden sind. Dieses Verfahren steht also im Einklang mit der geforderten Eigenstabilität.

Wie kann eine solche isApplicationAlive()-Funktion aussehen?

An dieser Stelle kann der Entwickler des Applikations-Servers jede gewünschte Art der Funktionsprüfung implementieren, aus Perspektive des Virtuellen Servers interessiert nur der boolesche Rückgabewert. Natürlich erscheint es sinnvoll, die Funktion aus Anwendersicht zu testen: Ist die gewünschte Applikation unter „www.virtueller-server.org“ verfügbar und korrekt, oder nicht? Aber auch interne Überprüfungen, wie z.B. der Verfügbarkeit der Datenbank, sind denkbar. In unserem Testaufbau begnügen wir uns mit dem Abruf eines speziellen Templates vom virtuellen Server, das in Folge auf das Enthalten-, bzw. nicht-Enthaltensein bestimmter Wörter untersucht wird. Auf diesem Wege wird die Dispatchingfaehigkeit des aktuellen Dispatchers, sowie die Funktion eines einzelnen Servers geprüft. Sollte letzterer und nicht der Dispatcher selbst für einen negativen Ausgang verantwortlich sein, so kann dies dennoch zu einer Neuwahl des Dispatchers führen (wenn bereits genug Fehlermeldungen eingegangen sind). Dieses Verhalten ist jedoch erwünscht, da es auf eine falsche Verwaltung der freien Server, bzw. auf falsche Annahmen über die Last und Verfügbarkeit dieser seitens des Dispatchers hindeutet.

3.2.2.3 Das Arp-Spoofing-Modul

Angenommen, eine Neuwahl des Dispatchers wurde eingeleitet, die Mehrzahl der teilnehmenden Server beginnt, Heartbeat-Meldungen zu versenden und stellt den Versand rasch wieder ein, nachdem ein Server höherer Priorität eine adäquate Heartbeat-Meldung verschickt hat. Diesen einzig weitersendenden Server gilt es nun, dem Router-/Firewallsystem als neuen Dispatcher anzumelden.

Unser Ziel ist es, einen Mechanismus für diesen Zweck zu nutzen, der eine Modifikation der Router-/Firewallsoftware nicht erfordert. Ein proprietäres Protokoll zur Bestimmung des Dispatchers kommt also nicht in Frage.

Wir haben uns entschlossen ein Verfahren zu nutzen, das im allgemeinen mit dem weniger seriös klingenden Namen „Arp-Spoofing“ bezeichnet

wird. Letztendlich verbirgt sich hinter diesem Verfahren lediglich die Nutzung eines Seiteneffektes des in [Plum82] vorgestellten Address Resolution Protocols.

Das Address Resolution Protocol, kurz ARP, dient der Abbildung von Protokolladressen (z.B. IP-Adressen) auf Hardwareadressen (z.B. MAC-Adressen). Möchte das IP-Modul ein Paket an eine andere IP-Adresse senden, so versieht es das Paket mit der gewünschten Zieladresse und gibt es weiter an das ARP-Modul. Dieses nimmt das Paket entgegen, liest die Zieladresse (in Form einer IP-Adresse) aus und übersetzt diese in die zugehörige Hardwareadresse, an die es das Paket in Folge weiterleitet. Dabei nutzt das ARP-Modul das ARP-Protokoll zur Identifizierung der zugehörigen Hardwareadresse: Ist ihm die Zieladresse noch unbekannt verwirft es zunächst das IP-Paket (in der Erwartung, dass es ein weiteres Mal gesendet werden wird) und sendet dann ein ARP-Packet des Typs ARP-Request aus. Dieser ARP-Request ist ein Broadcast an alle erreichbaren Hosts und beinhaltet die gesuchte Protokoll-Adresse, sowie den Urheber der Anfrage. Erreicht dieses Paket den Ziel-Host, merkt sich dieser die Protokolladresse des Urhebers und seine zugehörige Hardwareadresse – beides ist ebenfalls im ARP-Packet enthalten – und sendet selbst ein ARP-Packet des Typs ARP-Reply, die eigene Hardwareadresse enthaltend, zurück an den Initiator, bei diesem Paket handelt es sich um ein Gerichtetes. Der ursprünglich nachfragende Host wird dieses Paket erhalten, das enthaltene Paar aus Protokoll- und Hardwareadresse speichern und kann in Folge das nächste IP-Paket für diese Protokolladresse an die entsprechende Hardwareadresse schicken. Die Datenstruktur zur Speicherung dieser Paare wird mit ARP-Cache bezeichnet. Für unsere Zwecke wichtig ist, dass es sich bei dieser Kommunikation um ein verbindungsloses Protokoll handelt: Das anfragende ARP-Modul wartet nicht explizit auf eine Antwort, vielmehr nimmt jedes ARP-Modul Antworten (also ARP-Reply-Packets) entgegen und sendet im Bedarfsfall selbst Anfragen (ARP-Request-Packets) aus. An dieser Stelle setzt das Verfahren des ARP-Spoofings an: Nur wenige, besonders auf Sicherheit bedachte Router-/Firewallsysteme besitzen ein erweitertes ARP-Modul, das ARP-Replies lediglich dann bearbeitet, wenn es zuvor eine entsprechende Anfrage versandt hat. Die Mehrzahl der gängigen Router-/Firewall-Lösungen verfährt jedoch auf die nicht prüfende Weise, wie sie oben beschrieben wurde: Erhält das ARP-Modul ein ARP-Reply-Packet, so wird dieses bearbeitet, indem das enthaltene Paar aus Protokoll- und Hardwareadresse in den ARP-Cache übertragen wird, oder, sollte ein Eintrag zu eben dieser Protokolladresse bereits bestehen, dieser ein Update erfährt. (Die Aussage, dass es sich hier um die Mehrheit handelt, ist eine rein empirisch überprüfte These, die [Volo97] entnommen worden ist.) Eine Überprüfung auf vorhergegangene Anforderung dieser Antwort wird nicht vollzogen.

Wie lässt sich dieses Verhalten für unseren Zweck, dem Router-/Firewallsystem einen neuen Dispatcher zu verkünden, nutzen?

Alle Teilnehmer des virtuellen Servers besitzen zusätzlich zu ihrer individuellen IP-Adresse eine weitere virtuelle, die einheitlich ist und die die von außen zugängliche Adresse des virtuellen Servers darstellt. (Es ist dafür

zu sorgen, dass Überwachungsmechanismen zum Aufspüren mehrfach genutzter IP-Adressen für diese virtuellen Netzwerkgeräte deaktiviert sind.) Wenn nun eine Anfrage nach dieser IP-Adresse beim Router-/Firewallsystem eingeht, wird sie an dessen ARP-Modul weitergereicht, welches in seinem ARP-Cache nachschaut, ob die gewünschte Adresse bereits enthalten ist. Sollte dies der Fall sein, wird das Paket entsprechend weitergeleitet. An dieser Stelle möchten wir, dass die Anfragen genau den aktuellen Dispatcher erreichen. Sollte aber noch kein Eintrag bzgl. der Adresse des virtuellen Servers im ARP-Cache enthalten sein, würde das ARP-Modul ein entsprechendes ARP-Request-Packet aussenden. Eine sogenannte Race-Condition wäre geschaffen, bei der es um die zeitliche Reihenfolge geht, in der die Server auf diese Anfrage reagieren. Da alle Server die erfragte Protokolladresse für ihre eigene hielten, antworteten sie auch alle auf die Anfrage. Es wäre nicht sicherzustellen und auch nicht sehr wahrscheinlich, dass ausgerechnet der Dispatcher dieses „Rennen" gewönne. Der virtuelle Server, oder besser seine Teilnehmer müssen also von vornherein das Versenden von ARP-Anfragen nach der Hardwareadresse des virtuellen Servers unterbinden, bzw. nicht nötig werden lassen. Aus diesem Grund werden in regelmäßigen Intervallen ARP-Reply-Packets, die das Paar (Protokolladresse des virtuellen Servers, Hardwareadresse des aktuellen Dispatchers) vermitteln, an das Router-/Firewallsystem gerichtet. Periodisch muss dies geschehen, da bei vielen Implementierungen des ARP-Moduls die Einträge des ARP-Caches nach einiger Zeit entfernt werden.

Welcher Teil des Virtuellen Servers übernimmt das ARP-Spoofing ?

Es wäre natürlich denkbar, den Dispatcher selbst diese Reply-Packets senden zu lassen, schließlich kennt er die eigene Hardwareadresse am ehesten. Wie sich allerdings während der Implementierung herausstellte, ist dies nur bei Solaris der Fall (Die MAC-Adresse befindet sich als permanenter Eintrag im ARP-Cache). Unter Win32 und Linux ist bereits erheblich mehr Aufwand von Nöten. Zudem widerspricht es unserem Prinzip der Mehrheitsentscheidung, einen einzelnen Server entscheiden zu lassen, welcher Server der Dispatcher sei. Was wäre, wenn z.B. die MAC-Adresse, die der Dispatcher für die eigene hält, die falsche wäre? Keine einzige Anfrage erreichte ihn. (Natürlich würde dieser Fehler bei geeigneter Implementierung der isApplicationAlive()-Funktion behandelt werden, doch einige Zeit wäre der virtuelle Server nicht zu erreichen.) Diese Überlegungen führen zu dem Vorgehen, dass jeder teilnehmende Server bis auf den Dispatcher selbst, entsprechende ARP-Replies versendet; das Address Resolution Protocol erlaubt das Versenden von ARP-Reply-Packets „für einen anderen Host". Ferner senden alle Server mit unterschiedlichem zeitlichen Offset (das sich aus deren eindeutiger Kennzahl herleitet) diese ARP-Replies, was einen in einem Intervall möglichst gleichverteilten Versand dieser erwirkt. Es ist, ohne Modifikation der Software seitens Router/Firewall, nicht in vertretbarem Rahmen möglich, einzelne Server am Versand von ARP-Replies bezüglich falscher Inhalte zu hindern. Durch die zeitliche Gleichverteilung wird aber sichergestellt, angenommen ein Server geriete in der Praxis in den pathologischen Zustand, die falsche Hardwareadresse verschicken zu wollen, dass der virtuelle Server nur in einem Bruchteil der Zeit, gegeben

durch die Anzahl teilnehmender Server, nicht zu erreichen ist. Wie sich herausstellte, sind der Versand von ARP-Packets sowie die Bestimmung von eigener und fremder Ethernetadresse in der Art der Implementierung recht verwandt, so dass dieses Vorgehen keinen weiteren Implementierungsaufwand und somit kein unnötiges Aufblähen der notwendigen, nativen sendARP-Bibliothek unseres virtuellen Servers zur Folge hat. Wir nutzen für diese Zwecke unter Linux die socket(SOCK_PACKET)-, bzw. socket(DGRAM) + ioctl(SIOCGARP)-Schnittstelle, unter Win32 die freie Bibliothek [WinP], die Linux vergleichbare Schnittstellen bereitstellt. Um die eigene MAC-Adresse zu erfahren wird unter Win32 die microsofteigene IPHLPapi genutzt. Solaris stellt für den Versand von ARP-Packets leider keine geeignete Schnittstelle zur Verfügung, sodass es geplant ist, den Umweg über die DLPI-Schnittstelle zu gehen, was jedoch nicht mehr im Rahmen dieses Projekts geschehen soll. Eine teilnehmende Solaris-Maschine ist also derzeit nicht in der Lage, an der Verkündung des Wahlergebnisses teilzuhaben. Die Bestimmung der MAC-Adressen funktioniert unter Solaris analog zu Linux. Wie die Schnittstellen im einzelnen verwendet werden und das genaue Format der ARP-Packets ist in der Dokumentation der Sourcen nachzulesen.

4 Implementierung und Test

4.1 Details

Die für Lastverteilung und Ausfallsicherheit wesentlichen Klassen:

Sacristan

Das Herzstück der Lastverteilung des Pope ist eine Instanz der Klasse „Sacristan“. Die Funktionen werden in Abbildung 4 schematisch gezeigt für einen Sacristan, der vier Pews plus die Pew 0 verwaltet. Im Folgenden soll auf dessen Arbeitsweise im Einzelnen eingegangen werden.

Da wäre zum Einen bei Eintreffen einer initialen http-Anfrage das Auswählen des Priests, der ihr zugeordnet wird über dessen Repräsentanten in den Pews, den Confessor, und dessen Neueinordnung in das Pew-System. Dies zeigt das Aktivitätsdiagramm in Abbildung 5.

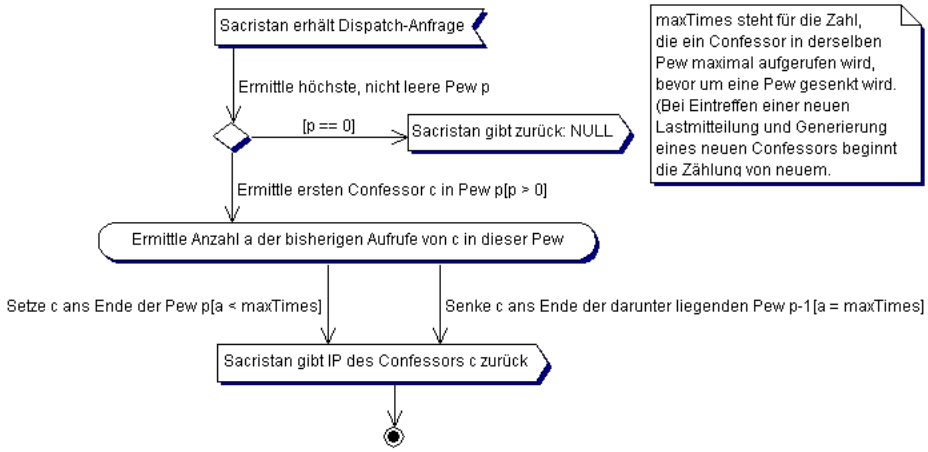


Abb. 5: Aktivitätsdiagramm der Lastverteilung

Wie die Confessor-Repräsentation bei Eintreffen neuer Lastmitteilungen eines Priests in den Pews aktualisiert wird, gibt Abbildung 6 an.

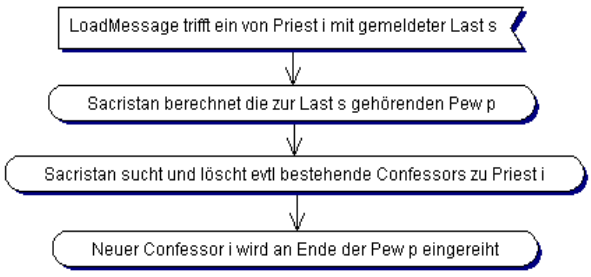


Abb. 6: Aktivitätsdiagramm der Pew-Zuordnung

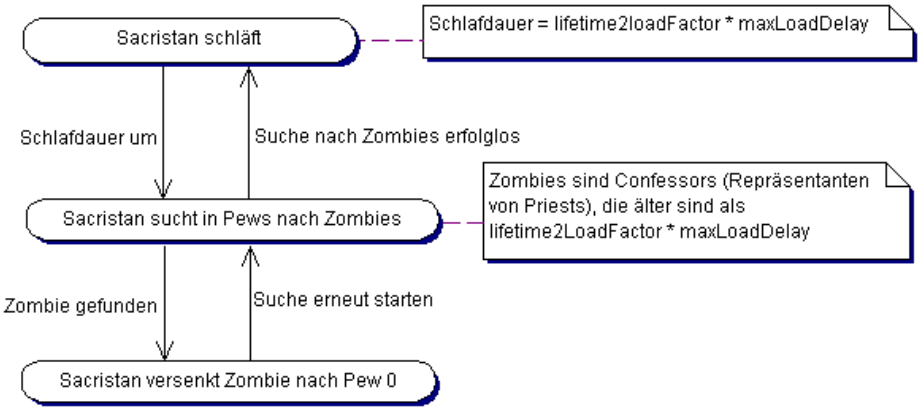


Abb. 7 beschreibt, wie Confessors, zu denen keine neuen Lastmitteilungen mehr eingehen, in Pew 0 versenkt werden. Dort verbleiben sie zu Debug-Zwecken.

Die dynamische Zuordnung von Lastzahlen auf die Pews geschieht nach dem Algorithmus von Abbildung 8.

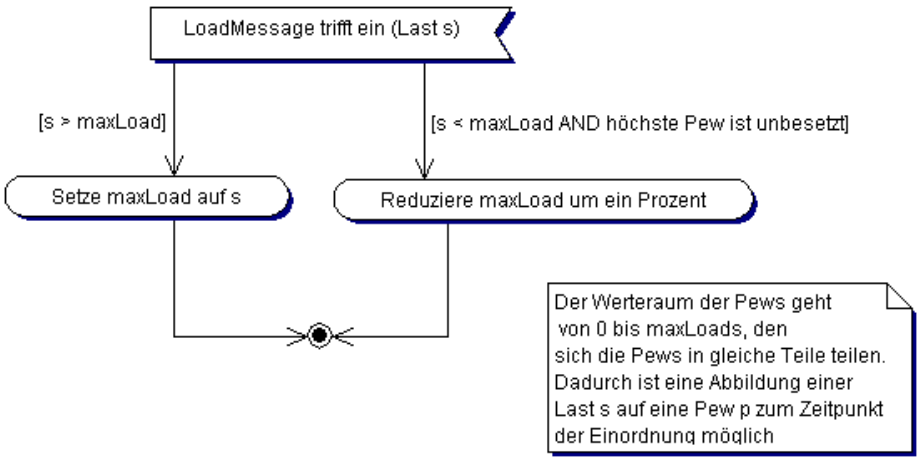


Abb. 8: Aktivitätsdiagramm zeigt die dynamische Berechnung der Werteräume der Pews

4.1.1.1 LoadTrader

Das Gegenstück zum Sacristan des Popes ist der LoadTrader-Thread auf den Priests. Tatsächlich existieren beide natürlich auf jedem Rechner, und der Sacristan ist sogar ein Teil des LoadTrader-Threads, doch das ist für den logischen Algorithmus unerheblich, sondern nur Bedingung dafür, dass jeder Priest den Pope ersetzen kann. Der LoadTrader versendet die Lastmitteilung mit der aktuellen Lastkennzahl, die er von der mit ihm verbundenen Software erfährt. Die Arbeitsweise des LoadTraders, die im folgenden Zustandsdiagramm in Abbildung 9 beschrieben wird, stellt sicher, dass das Intervall zwischen zwei versandten Nachrichten ein festes Zeitlimit nicht überschreitet.

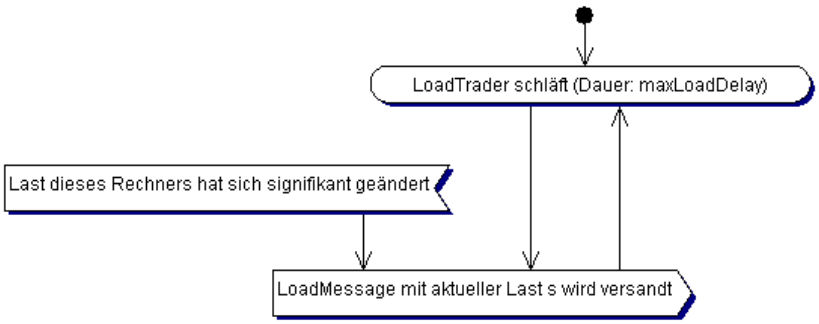


Abb. 9: Lastmitteilung folgt sowohl zeit-, wie auch ereignisabhängig.

4.1.2 Ausfallsicherheit

4.1.2.1 Heartbeat-Sender

Schaltet der IPvise in den Pope-Modus, wird der Heartbeat-Sender aktiviert und sendet von nun an jede Tatkteinheit eine mit der zum Zeitpunkt seiner Aktivierung gültigen akzeptierten Kennung versehene Heartbeat-Meldung, bis der IPvise den Pope-Modus wieder verlässt und den Heartbeat-Sender deaktiviert.

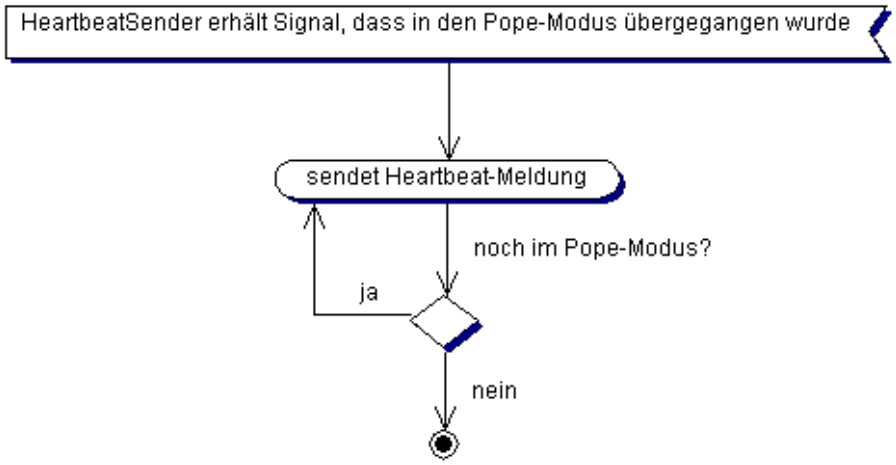


Abb.10: Aktivitätsdiagramm des Heartbeat-Versands

4.1.2.2 Heartbeat-Counter

Erhält ein Server über k Takte hinweg keine gültige Heartbeat-Meldung, initiiert der Heartbeat-Counter eine Neuwahl des Popes indem er die akzeptierte Kennung inkrementiert und den IPvise in den Pope-Modus überführt.

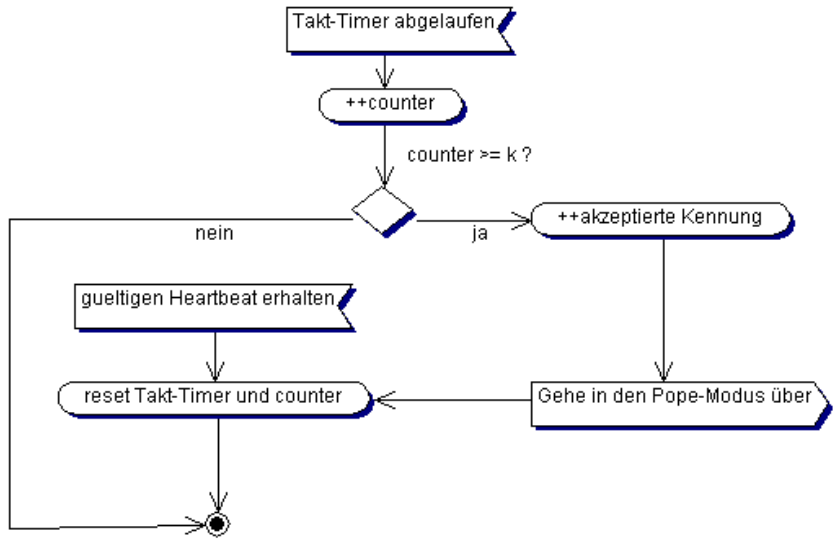


Abb. 11: Aktivitätsdiagramm des Überganges eines Priests in den Popestatus

4.1.2.3 Heartbeat-Message-Handler

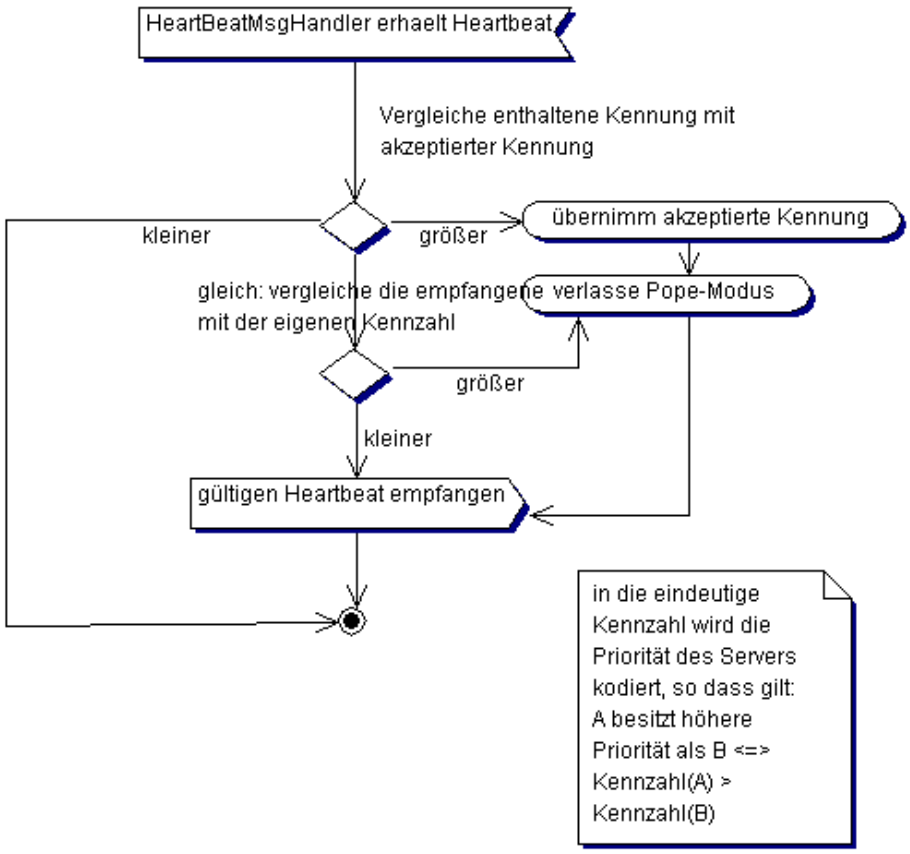


Abb. 12: Aktivitätsdiagramm der Auswertung eines Heartbeats

Eingehende Heartbeats überprüft der Heartbeat-Message-Handler auf ihre Gültigkeit hin, benachrichtigt gegebenenfalls den Heartbeat-Counter und kann die Teilnahme an der Wahl zum Pope zurückziehen, falls er die Heartbeat-Meldung eines Servers höherer Priorität erhalten hat (Abb 12).

4.1.3 Schnittstellen

Um den IPvise mit einer Software, typischerweise einer Webapplikation, zu verknüpfen und diese damit auf verteilten Rechnern laufen zu lassen, bedarf es einer Schnittstelle zwischen den beiden. Der IPvise ist ein Java-Paket „ipvise“, das importiert und über die Instantiierung der Klasse „IPvise“ aktiviert wird. Im Folgenden werden die bereitgestellten und erwarteten Methoden aufgeführt.

4.1.3.1 IPviseInterface

Auf Seiten des virtuellen Servers wird erwartet, dass die verknüpfte Applikation (wie erwähnt, im Prototyp der WebMediator) über die Schnittstelle „IpviseInterface“ drei Methoden bereitstellt: Ein Objekt der Implementierung der Schnittstellenklasse wird der Klasse IPvise bei Instantiierung übergeben.

```
int getMyLoad()
```

Gibt die aktuelle freie Kapazität des Rechners abgebildet auf eine positive Ganzzahl zurück. Welche Betriebsmittel die Applikation bei dieser Erhebung miteinbezieht, und wie sie deren Auslastung gewichtet, ist dabei – sinnvollerweise – ihre Entscheidung. Der IPvise erwartet nur, dass die Kapazität desto höher ist, je höher die gemeldete Kennzahl ist.

```
boolean: isApplicationAlive(String: actualPope)
```

Stößt einen Test der Applikation und des Popes „von außen" an. Soll Funktionsstörungen des Pope erkennen, wenn er nichtsdestotrotz noch in der Lage ist, Heartbeats auszusenden. Der Übergabeparameter ist die IP-Adresse des aktuellen Popes.

```
void: SendTrap(int defcom, String type, String msg)
```

Ermöglicht Logging, Ausgaben und Fehlermeldungen, wobei die Applikation über dem IPvise entscheiden kann, ob und wie sie diese behandelt. Der Parameter „defcom" zeigt die Dringlichkeit, „type" die Kategorie, und „msg" ist der Inhalt der Nachricht.

4.1.3.2 IPvise

Demgegenüber bietet die Klasse IPvise der Applikation die folgenden Methoden.

String: nominateServer()

Liefert die IP-Adresse des Rechners (Priests) im Netz, der die Abarbeitung, der nächsten Session übernehmen soll.

void: transmitLoad()

Übermittelt sofort die aktuelle freie Kapazität dieses Rechners an den Pope. Dies ist dann wichtig, wenn die Applikation eine signifikante Änderung der Last festgestellt hat, und nicht warten will, bis die nächste turnusgemäße Lastübermittlung stattfindet. Es steht der Applikation frei, diese Funktionalität zu nutzen oder eben nicht.

Altarpiece: getBalancingSnapshot()

Zu Überwachungs- und Debugzwecken gibt diese Methode die Belegung der eigenen Pew-Struktur mit Confessors zurück. Altarpiece ist eine Spezialisierung der Vector-Klasse. Das Altarpiece liefert auch Informationen über einige der wichtigsten Parameter der Priests zurück, die diese mit der Lastmitteilung versandt haben.

4.1.3.3 Parameter

Darüberhinaus bietet die Klasse IPvise Methoden zur Ausgabe von Variablenwerten, oder Methoden, um Variablen nach Instantiierung des IPvise nachträglich zu verändern. Details dazu finden sich in der Dokumentation des Programmcodes.

4.2 Test

4.2.1 Szenario

Für die Tests standen fünf Rechner zur Verfügung: Nachdem Solaris nicht in das IPvise-System zu integrieren war, nur noch drei statt der ursprünglich geplanten vier Clusterrechner. Alle mit „Family 6 Model 8 Step 10 GenuineIntel“ 1000-Mhz-Prozessoren, 384 MB Arbeitsspeicher, von denen zwei mit Windows 2000 (Service Pack 2) und einer mit einer Suse 7.3 Linux-Distribution als Betriebssystemen ausgestattet worden sind. Ein weiterer Linux-Rechner fungierte als Router, ein anderer Linux-Rechner trug die Datenbank für die mit dem WebMediator realisierten Applikation.

Auf den Cluster-Rechnern war Apache als Web-Server im Einsatz.

4.2.2 Ausfallsicherheit

Ist der Pope heruntergefahren worden, oder wurde er einfach vom Netzwerkkabel abgehängt, hat binnen weniger Sekunden einer der beiden verbleibenden Priests die neue Pope-Rolle bekommen. Eventuelle Requests, die in der Zwischenzeit an die Adresse des virtuellen Servers gingen, wurden nicht bedient. War der ursprüngliche Pope nur vom Netz genommen, aber hat weiterhin seine Rolle ausgeführt, so erkannte er wenige Sekunden nach Wiederanschluss an das Netz den neuen Pope an, und übernahm die Rolle eines Priests. Ob in der kurzen Zeit, in der zwei Popes am Netz waren, Requests an den „falschen“ gelenkt worden sind, war nicht zu beobachten. Dies wäre aber bedeutungslos, da sie genauso bedient und verteilt worden wären.

Insofern wird die Funktionalität „Ausfallsicherheit“ bezüglich des Dispatchers gewährleistet.

4.2.3 Lastverteilung

Einen ersten Eindruck von der Funktionstüchtigkeit in punkto Lastverteilung ergab ein einfacher Test. Wurde mehrmals die Applikation über die Adresse des virtuellen Servers im Browser aufgerufen, erschien in der URL-Zeile wechselnd die Adresse der beiden Priests.

Die Güte und Leistungsfähigkeit des Verteil-Algorithmus zu testen führt nur zu wirklich verwertbaren Aussagen, wenn zum einen auf Seite der Software (hier des WebMediators) ein vernünftiger Mechanismus zur Ermittlung der Lastverteilung implementiert worden ist, und zum anderen die Leistungsfähigkeit der Clusterrechner stark abweicht. Beides ist in unserem Testszenario noch nicht der Fall: Die Rechner sind aus der gleichen Serie, und für den Algorithmus werden zwar schon Strategien entwickelt, aber bis er implementiert worden ist, gibt es nur einen simplen Workaround. So, dass die Lastverteilung bis dato gar nicht ihre vollen Fähigkeiten ausspielen kann, sondern wie Round-Robin wirkt.

4.2.4 Pope

Für jedes System, das unter großer Last stehen wird, ist immer die Frage nach dem schwächsten Glied wichtig, dem Flaschenhals. Dabei kann man den IPvise nicht singulär betrachten, da sich auch andere Komponenten direkt auf die Effizienz auswirken. Da wäre zum einen der Web-Server und dessen Konfiguration zu nennen. Entscheidend ist auch die Art des Netzes, das die Cluster-Rechner verbindet oder die Bandbreite der Anbindung an das Internet. Einschränkungen können sich auch ergeben durch die Leistungsfähigkeit des WebMediators, die Anforderungen der Appliktionen, die darauf verwirklicht worden sind und natürlich durch deren Datenbank-Anbindungen.

Eine wichtige Messgröße ist dabei die Anzahl der Requests pro Zeiteinheit, die das System bewältigen kann. Nach Erfahrungswerten der Firma IconParc kann ein Rechner mit der Leistungsfähigkeit der im Szenario vorgesehen Clusterrechner und einem auf dem WebMediator basierenden Web-Shop darauf etwa zehn Requests pro Sekunde verarbeiten. Im Durchschnitt könne man außerdem von etwa 15 Requests pro Session ausgehen. Dabei ist zu beachten, dass hier von einem Mittelwert der Last ausgegangen wird, die ein Request (im Folgenden auch „Klick“) erzeugt, da jeder Request entweder eine kurze statische Textantwort, oder auch eine umfangreiche Datenbanktransaktion auslösen kann.

Durch den Einsatz eines virtuellen Servers kann (innerhalb der Grenzen, die das Netz dem Durchsatz setzt) diese Anzahl Klicks prinzipiell durch den Einsatz von zusätzlichen Rechnern im Verbund erhöht werden, abgesehen vom Pope, über den alle initialen Requests laufen müssen. Also haben wir die maximale Leistungsfähigkeits des Popes einzeln betrachtet. Das dafür eingesetzte Werkzeug war der Javabasierte Jmeter [Jmet] von „The Java Apache Project“, der automatisch Requests an eine http-Adresse sendet und deren Antwortzeiten misst. Dazu können gleichzeitig bis zu 20 Threads geöffnet werden. Im Folgenden die Tabelle mit den Testergebnissen.

Anzahl Threads, die gleichzeitig Requests senden	Testdauer in Sekunden	Anzahl abgesandter Requests	Minimale Antwortzeit in ms	Maximale Antwortzeit in ms	Mittlere Antwortzeit in ms	Requests pro Sekunde (berechnet nach...)	
						...Anzahl Requests	...mittlere Antwortzeit
1	60	1174	40,0	441,0	46,9	19,6	21,3
2	60	1305	40,0	3044,0	87,1	21,8	23,0
5	60	1323	40,0	3194,0	220,1	22,1	22,7
10	60	1326	40,0	3956,0	444,5	22,1	22,5
15	60	1315	40,0	4487,0	675,0	21,9	22,2
20	60	1291	40,0	1922,0	912,7	21,5	21,9

Die Requests waren an die Adresse des Virtuellen Servers gerichtet, der die URL des Priests, an den er den Request weitergeleitet hätte, als Ascii-Text zurückgesandt hat. Die Requests pro Sekunde wurden daraus einmal berechnet auf Grundlage der Anzahl der Requests, die zur Testdauer abgesandt wurden, und einmal auf Grundlage der mittleren Antwortzeit.

Geht man also von einem Maximum von etwa 21 initialen Requests aus, die der Pope pro Sekunde verarbeiten kann, also 21 Sessions, die pro Sekunde eröffnet werden, so kann man unter der Annahme, dass eine Session aus etwa 15 Klicks besteht, abschätzen, dass die Priests hinter dem Pope auf maximal etwa 300 Klicks pro Sekunde vorbereitet sein müssen. Das würde mit Rechnern, die zehn Klicks pro Sekunde verarbeiten können, auf einen Cluster mit rund 30 Rechnern hinauslaufen. Wohlgemerkt, das sind alles nur grobe Abschätzungen und Annahmen, aber sie geben ein Gefühl für die Größenordnung der Rechnerverbünde, die mit dem IPvise möglich würden – vorbehaltlich anderer Beschränkungen bei Netzlast, Datenbankserver oder Ähnlichem.

5 Zusammenfassung

Der IPvise ist gemäß den Anforderungen entworfen und implementiert worden und erfüllt seinen Zweck in punkto Lastverteilung und Ausfallsicherheit hinlänglich. Kleinere Einschränkungen haben sich bei seiner Entwicklung ergeben. So können derzeit Solaris-Rechner im Netz zwar Last tragen, aber nie die Rolle des Dispatchers übernehmen, was einen Rechnercluster verbietet, der allein aus Solaris-Maschinen besteht. Zum anderen stößt auch der IPvise zahlenmäßig an eine Grenze, da der Broadcastverkehr des IPvise linear mit der Zahl der Rechner zunimmt. Es gibt aber bereits Konzepte, auch diese Grenze durch eine Hierarchie von Einzelnetzen zu überwinden. Da deren Verwirklichung den Rahmen eines Fortgeschrittenenpraktikums beziehungsweise Systementwicklungsprojektes gesprengt hätte, begnügen wir uns mit dem Wissen, dass auch sehr große Cluster mit der Grundstrategie des IPvise in den Griff zu bekommen sind.

Diese Einschränkungen beeinträchtigen den Betrieb des IPvise wohl wenig, was uns hoffen lässt, dass er auch im kommerziellen Einsatz bestehen wird.

Obwohl wir bei unserer Entwicklung von Ethernet als Kommunikationsplattform ausgegangen sind, ist der IPvise grundsätzlich auch auf einem peer-to-peer-Netz wie ATM lauffähig, das einen Broadcast simulieren kann. Dies bedarf aber Anpassungen bezüglich des Adresswechsels mittels ARP-Spoofing.

6 Literatur- und Softwareverzeichnis

6.1 Literatur

[Ludw96] Ludwig, Thomas: Lastverwaltung für Parallelrechner (Reihe Informatik Bd 94). BI-Wiss.-Verl. Mannheim; Leipzig; Wien; Zürich; 1993.

[Popi96] Popien, Claudia: Verteilte Systeme (Aachener Beiträge zur Informatik, Bd. 16) Verlag der Augustinus Buchhandlung, Aachen, 1996.

[Müll00] Müller, Andreas: Linux Clustering (Paper zu einem Vortrag an der Linux-Conference am 27. Juni 2000 in Zürich).

[Brüc02] Brückner, Roland: Whitepaper IconParc eBusiness Framework V4.0 (www.iconparc.de), 2000.

[Plum82] Plummer, David C.: Request for Comments 826 - An Ethernet Address Resolution Protocol (www.faqs.org/ rfcs/rfc826.html), 1982.

[Volo97] Volobuev,Yuri: ARP and ICMP redirection games (www.insecure.org/sploits/arp.games.html), 1997.

6.2 Software

[WinP] WinPcap v2.02: the Free Packet Capture Architecture for Windows, (winpcap.polito.it).

[JMet] Apache Jmeter v1.4.5-dev: The Jakarta Project, (jakarta.apache.org/jmeter).

7 Abbildungsverzeichnis

Architekturentwurf 1: Ein ausgezeichneter Rechner (Pope) übernimmt allein die Verteilung. Die anderen Rechner (Priests) erhalten die Sessions. – Seite 11

Architekturentwurf 2: Die Zustandsüberwachung des virtuellen Servers ermöglicht ein Event-/Polling-basiertes Mischmodell. – Seite 14

Abb. 1: Konzept des virtuellen Servers – Seite 4

Abb. 2: Arbeitsweise des WebMediators nach [Brüc02] – Seite 5

Abb. 3: Phasen der Lastverteilung nach [Ludw96] – Seite 8

Abb. 4: Dynamische Multilevel Feedback Queues – Seite 13

Abb. 5: Aktivitätsdiagramm der Lastverteilung – Seite 21

Abb. 6: Aktivitätsdiagramm der Pew-Zuordnung – Seite 21

Abb. 7 beschreibt, wie Confessors, zu denen keine neuen Lastmitteilungen mehr eingehen, in Pew 0 versenkt werden. Dort verbleiben sie zu Debug-Zwecken. – Seite 22

Abb. 8: Aktivitätsdiagramm zeigt die dynamische Berechnung der Werteräume der Pews – Seite 22

Abb. 9: Lastmitteilung folgt sowohl zeit-, wie auch ereignisabhängig. – Seite 23

Abb.10: Aktivitätsdiagramm des Heartbeat-Versands – Seite 24

Abb. 11: Aktivitätsdiagramm des Überganges eines Priests in den Popestatus – Seite 24

Abb. 12: Aktivitätsdiagramm der Auswertung eines Heartbeats – Seite 25