Plug&Participate für „limited devices“ mittels Gateway-Strukturen

Steffen Deter
Philipps-Universität Marburg
Fachbereich Mathematik and Informatik
deter@mathematik.uni-marburg.de

Abstract

Plug&Play-Technologien ermöglichen im allgemeinen das Einfügen von Komponenten in ein System und die darauffolgende Nutzung dieser Komponenten. Im Bereich der Netzwerktechnologie und der Verteilten Systeme existieren ebenfalls solche Plug&Participate-Technologien, die unter dem Begriff “Middleware” zusammengefaßt werden. Zu diesen Middleware-Technologien gehört unter anderem die Jini-Technologie, welche von der Firma Sun Microsystems entwickelt wurde. Primär wurde Jini mit dem Ziel entwickelt, auf Geräten mit geringer Ressourcenausstattung eingesetzt zu werden. Dies ist bis heute nicht vollständig gelungen. Dieser Artikel wird neben einer kurzen Einführung in die Jini Technologie und der Herausarbeitung verschiedener Nachteile und Schwachstellen, eine Plug&Participate Technologie vorstellen, welche speziell für den Einsatz auf genannten Geräten entwickelt wurde. Weiterhin wird dieser Artikel auf die Gemeinsamkeiten und Unterschiede beider Plug& Participate Technologien eingehen und im Anschluß daran eine Gateway Architektur vorstellen, welche es in einfachster Weise ermöglicht, „limited devices“ in Jini Gemeinschaften aufzunehmen. Zusätzlich wird diese Gateway-Struktur um eine Lastbalancierungskomponente erweitert, die es ermöglichen wird, daß die in einem System vorhandenen JP-Gateways gleichermaßen ausgelastet werden.

Motivation

In heutiger Zeit gewinnen Geräte wie Mobiltelephone, PDAs, Handhelds, etc. immer mehr an Bedeutung. Sie stellen zum Teil eine Grundvoraussetzung für Mobilität und Kommunikationsmöglichkeiten bereit. Solche Geräte haben insbesondere die Gemeinsamkeit, daß sie in die Klasse der sogenannten „limited devices“ fallen. Es gilt somit, daß die Geräte zumeist nur beschränkte Speicher- und Rechenkapazitäten aufweisen und oftmals auch nur bedingte graphische Ausgaben ermöglichen.
Aber nicht nur im alltäglichen Leben gewinnen diese Geräte immer mehr an Bedeutung, sondern sie sind auch verstärkt im Bereich der Fabrikautomatisierung zu finden. Dennoch gilt hier, daß die Automatisierungsgeräte eine gewisse Form an „Intelligenz“ im Sinne von Speicher- und Rechenressourcen besitzen. Genau diese „Intelligenz“ soll im Rahmen des PABADIS-Projektes stärker ausgenutzt werden, indem flexible und anpassungsfähige Strukturen durch die Kombination von Plug&Participate Technologien und Agententechnologien bereitgestellt werden (siehe hierzu [6]). Die Forderung nach solch flexiblen und anpassungsfähigen Strukturen ergibt sich aus der Dynamik der heutigen Märkte. Zudem muß insbesondere im Bereich der industriellen Automatisierung die Eigenschaft der Plattformunabhängigkeit stark berücksichtigt werden, da hier eine breitgestreute Menge unterschiedlichster Plattformen existiert. Aus diesem Grund ist als Plug&Participate System die Jini-Technologie zu bevorzugen, da sie vollständig auf der plattformunabhängigen Programmiersprache Java basiert. [1][2] Allerdings ergibt sich hier ein Problem: Obwohl Jini von Sun Microsystems mit dem Ziel entwickelt wurde, auf „limited devices“ eingesetzt zu werden, ist dieses Ziel bisher nicht verwirklicht worden.
Um dennoch ein Jini-ähnliches Plug&Participate bereitstellen zu können, wurde an der Philipps-Universität Marburg eine (Jini-ähnliche) Plug&Participate Technologie namens Pini entwickelt. [5] Diese stellt eine eigenständige Plug&Participate Welt für „limited devices“ bereit, welche zudem über eine Gatewaytechnologie an Jini Gemeinschaften problemlos angebunden werden kann. Dieser Gateway soll neben einer knappen Einführung in die genannten Plug&Participate Technologien das Zentrum dieses Artikels darstellen. Schließlich werden Erfahrungen mit dem Gateway dargelegt und eine mögliche Erweiterung dieser Architektur im Hinblick auf Lastbalancierung vorgestellt.

Jini und Pini – ähnliche, dennoch gegensätzliche Plug&Participate Technologien

Dieser Abschnitt wird zwei Plug&Participate Technologien vorstellen, welche einander sehr ähnlich sind, aber dennoch für gegensätzliche Plattformen designed wurden. Beide Technologien, Jini und Pini, basieren grundlegend auf der gleichen Architektur: *Services*, *Service Users* (=Clients) und einem speziellen Service, genannt *Lookup Service*, welcher grundsätzlich als zentrale Registrierungsinstanz für Services arbeitet. [1][2][5] Insbesondere gilt nämlich, daß beide Plug&Participate Technologien Infrastrukturen, bestehend aus Javaklassen und –interfaces, für

den Aufbau von ad-hoc Netzwerken bereitstellen. Weiterhin weisen beide Technologien folgende Features auf: Discovery, Join, Lookup, Leasing, Remote Events und Attribute.
Diese Komponenten sollen im folgenden für die beiden Technologien vorgestellt und verglichen werden.

Discovery

Diese Komponente dient dazu, in einer Gemeinschaft Lookup Services zu kontaktieren. Das *Discovery*protokoll besteht aus drei Einzelprotokollen, wovon zwei keine Kenntnis von der zugrundeliegenden Infrastruktur verlangen und das verbleibende Protokoll eine solche Kenntnis voraussetzt. Letztgenanntes Protokoll ist das Unicast-Discoveryprotokoll, mit dessen Hilfe bereits bekannte Lookup Services direkt kontaktiert werden können. Im Gegensatz dazu ist bei den beiden anderen Protokollen kein derartiges Wissen notwendig: In Jini werden Multicast-Discoveryrequests versendet, welche von Lookup Services empfangen und in entsprechender Weise beantwortet werden. In Pini dagegen wird statt Multicast Kommunikation Broadcast verwendet, da auf „limited device" Plattformen wie der KVM/CLDC (Kilo Virtual Machine/Connected Limited Device Configuration [7]) oftmals keine Multicastkommunikation unterstützt wird.
Das gemeinsame Resultat der Jini und Pini Discoveryprotokolle ist die Bereitstellung von Softwareobjekten, welche das Interface `ServiceRegistrar` implementieren: In Jini wird dieses Objekt als `java.rmi.MarshalledObject` übertragen, per dynamischem *Remote Class Loading* [3] die zugehörige Klassendatei geladen und dann das initialisierte Objekt an das anfragende Entity übergeben. Dieses Softwareobjekt wird (Lookup) Service Proxy genannt. Im Gegensatz dazu wird in Pini lediglich eine `ServiceDescription` vom Lookup Service an die anfragenden Entities gesendet. Diese können dann mittels dieser Informationen die bereits vorliegenden Proxystubklassen initialisieren. (Zu Proxystubs siehe folgenden Abschnitt „Join und Lookup")

Join und Lookup

Damit Services in den Plug&Participate Gemeinschaften sichtbar und damit auch nutzbar sind, müssen sie sich bei den entsprechenden Lookup Services anmelden. Hierzu dient das Join Protokoll, welches in Jini und Pini in gleicher Weise verwendet wird. Allerdings beruhen Jini und Pini *Join Protokoll* auf unterschiedlichen Prinzipien. Grundlegend gemeinsam haben beide Plug&Participate Technologien, daß Services vor einer Registrierung bei einem Lookup Service zunächst mittels des entsprechenden Discoveryprotokolls **mindestens** einen Lookup Service kontaktieren müssen [1][2], um dessen Lookup Service Proxy zu erhalten bzw. den Lookup Service Proxystub initialisieren zu können. Anschließend wird durch den Aufruf der Methode `register(...)` des Proxys der Join Prozeß angestoßen. In Jini wird nun das Proxyobjekt des zu registrierenden Services als `java.rmi.MarshalledObject` an den Lookup Service gesendet.
In Pini dagegen wird aus dem Proxyobjekt des zu registrierenden Services eine `ServiceDescription` extrahiert [5], welche dann an den Lookup Service gesendet und dort gespeichert wird. Durch dieses Prinzip bauen Lookup Services ein Abbild der jeweiligen Gemeinschaft auf; sie wissen also, welche Services im jeweiligen Netzwerk verfügbar sind. Durch die Verwendung von `ServiceDescriptions` in der Pini Technologie anstelle von vollständigen Proxyobjekten wird zum einen die Netzwerklast gesenkt, zum anderen wird Speicherplatz beim Lookup Service gespart. Insbesondere kann durch die Verwendung der `ServiceDescription`-Objekt besser vorhergesagt werden, welchen Speicherplatz ein Service beim Lookup Service belegen wird.

Mittels des *Lookup Protokolls* können Clients nach Services suchen, die in den Plug&Participate Gemeinschaften registriert sind. Die Voraussetzung dafür ist wiederum die Kenntnis eines Lookup Services und damit auch das Vorhandensein eines Lookup Service Proxys bzw. Proxystubs. Der Lookup Prozeß wird durch den Aufruf einer entsprechenden `lookup(...)`-Methode des jeweiligen Proxys angestoßen. Als Resultat werden die Proxyobjekte gefundener Services geliefert: In Jini werden die Proxyobjekte als `java.rmi.MarshalledObjects` übertragen, der zugehörige Klassencode per dynamischem Remote Class Loading geladen und dann das initialisierte Proxyobjekt an den Client zurückgegeben. In Pini dagegen werden lediglich Informationen über den Service zum anfragenden Client gesendet, so daß dieser in der Lage ist, den entsprechenden Proxystub zu initialisieren. Für diesen Stub gilt, daß er bereits vorliegen muß und nicht erst mittels Remote Class Loading geladen wird. Diese Forderung wird wie folgt erfüllt: Da sowohl in Jini als auch in Pini das Interface des Services, der genutzt werden soll, vorliegen muß, wird mittels eines speziellen Compilers dieses Interface in eine Stubklasse umgewandelt. Dieser Stub entspricht im wesentlichen einem PiniRMI Stub und implementiert das entsprechende Service Interface.
Mittels dieses Proxystubs können Clients nun den entsprechenden Service nutzen. Allerdings gilt, daß die Servicefähigkeiten genau auf dieses Interface beschränkt bleiben. Außerdem gilt, daß die Nutzung von Services ausschließlich auf dem Aufruf entfernter Methoden (*Remote Method Calls*) basiert (vgl. Abschnitt „PiniRMI"). Hier ist die Jini Technologie eindeutig im Vorteil, da beliebig komplexe Proxystrukturen möglich sind. Insbesondere können die Proxies ihr eigenes GUI mitbringen. Allerdings wird diese „Freiheit" nur dadurch erkauft, daß

dynamisches Remote Class Loading möglich ist. Hierdurch könnte sich eine Lawine von Klassen ergeben, die geladen werden müssen. Denn es gilt, daß ein Proxy intern weitere Klassen referenzieren kann, welche wiederum andere Klassen benutzen können. Daraus folgt vorallem, daß eine enorme Menge Speicherplatz notwendig werden kann. Eben dieser Speicherplatz ist auf „limited devices“ nicht vorhanden. Insbesondere kann in oben beschriebenem Fall nicht vorhergesagt werden, welche Speichermenge der Service benötigen wird und ob diese Speicheranforderungen durch das „limited device“ erfüllt werden können.
Dieses Problem tritt in Pini nicht auf, da es zum einen speziell für den Einsatz auf „limited devices“ entwickelt wurde, zum anderen durch die Verwendung von Proxystubs im vorhinein **exakt** bestimmt werden kann, welche Ressourcenmenge ein Service beanspruchen wird.
Ein weiterer wesentlicher Unterschied zwischen Pini und Jini besteht in den benutzten Kommunikationsmechanismen bei der Servicebereitstellung/Servicebenutzung. Die in diesem Abschnitt erläuterten Protokolle basieren ausschließlich auf dem Aufruf entfernter Methoden. In Jini wird hierfür die Java RMI Technologie [1][2][4] verwendet. Dadurch wird allerdings ein enormer Ressourcenverbrauch hervorgerufen. Insbesondere gilt, daß diese benötigten Ressourcen auf „limited devices“ im allgemeinen nicht verfügbar sind. Aus diesem Grunde wird in Pini eine entsprechende Technologie verwendet, welche speziell für den Einsatz auf „limited devices“ entwickelt wurde und mit den knappen Ressourcen sparsam umgeht: PiniRMI. (Zu weiteren Detail zu PiniRMI siehe Abschnitt „PiniRMI“ und [5].)

Leasing und Remote Events

Mittels Leasing wird die Selfhealing Eigenschaft der beiden Plug&Participate Technologien realisiert. Hierbei gilt, daß Ressourcen lediglich für eine bestimmte Zeit (Leasingdauer) vergeben werden. Diese Leasingdauern müssen periodisch verlängert werden, sonst laufen die Leases ab, und die entsprechenden Services werden aus der Gemeinschaft ausgeschlossen.
Um diese Erneuerung zu erreichen, muß im entsprechenden Leaseobjekt die Methode `renew()` aufgerufen werden, welche eine entfernte Methode beim Leasinggeber (z.B. Lookup Service) aufruft. In Jini wird dieser entfernte Methodenaufruf mittels Java RMI, in Pini dagegen mittels PiniRMI erfolgen.
Mit Hilfe von Remote Events können Entities einer Plug&Participate Gemeinschaft über Ereignisse informiert werden, die irgendwo in der Gemeinschaft aufgetreten sind. Hierfür müssen sich die interessierten Entities bei entsprechenden Instanzen (z.B. Lookup Service) registrieren; die Information über ein Ereignis erfolgt dann durch einen entfernten Methodenaufruf bei den entsprechend registrierten `RemoteEventListener`-Objekten. In Jini erfolgt dieser Methodenaufruf (per Definition) mittels Java RMI, in Pini dagegen mittels PiniRMI.

PiniRMI im Überblick

PiniRMI ist eine Technologie zum Aufrufen von entfernten Methoden in **ressourcenkritischen** Umgebungen wie „limited devices“. Die Grundlage dieser Technologie ist ein in ein Stub/Skeleton-Modell verkapseltes *Remote Procedure Call* Protokoll (RPC, siehe zum verwendeten RPC Protokoll auch [5]). Die Verkapselung erfolgt zum einen, um dem bekannten objektorientierten Programmiermodell zu genügen, zum anderen, um Typsicherheit zu erreichen: Die Stubs und Skeletons werden mittels eines PiniRMIC-Compilers erzeugt, welcher zunächst den entsprechenden Java Quellcode generiert. Dieser wird anschließend durch den JDK-Compiler `javac` in Java-Bytecode compiliert.
Insbesondere gilt also, daß bei einem entfernten Methodenaufruf die jeweilige Methode des entsprechenden Stubobjektes aufgerufen wird. Innerhalb des Stubs wird dann ein RPC-Methodenaufrufrequest erzeugt, welcher dann zum entsprechenden Serverobjekt gesendet wird. Auf der Serviceseite wartet ein Skeletonobjekt auf eintreffende RPC-Methodenaufrufrequests, welche dann innerhalb des Skeleton ausgewertet werden, infolgedessen ein entsprechendes `MethodCallEvent` ausgelöst wird. Mittels eines `MethodCallEventHandlers` werden nun die eigentlichen Methodenaufrufe durchgeführt und die Ergebnisse an den anfragenden Client bzw. dessen Stub zurückgesendet. Für weitere Details zum PiniRMI Protokoll bzw. Details zum verwendeten RPC Protokoll siehe [5].
Ein wesentlicher Punkt im Vergleich zwischen PiniRMI und Java RMI ist die Codegröße: Java RMI benötigt mindestens 401 kB Speicherplatz, während für PiniRMI 29 kB Speicherplatz ausreicht. Allerdings muß auch erwähnt werden, daß PiniRMI im wesentlichen auf die Funktionalität des Aufrufens entfernter Methoden beschränkt ist, während die Java RMI Technologie weitere nützliche Features bereitstellt.
Des weiteren müssen die zur Laufzeit verbrauchten Ressourcen verglichen werden. Leider liegen hier keine genauen Messungen vor. Grundlegend gilt aber, daß PiniRMI auf der KVM/CLDC erfolgreich getestet wurde, somit insbesondere auch auf „limited devices“ lauffähig ist. Im Gegensatz dazu gilt, daß Java RMI auf „limited devices“ nicht lauffähig ist. Zudem ergaben Laufzeitmessung zwischen Pini und Jini einen um Faktor 10 geringeren Speicherverbrauch.

Eine Gateway Technologie für Jini-Plug&Participate von „limited devices“

Nachdem im vorhergehenden Abschnitt zwei Plug&Participate Technologien vorgestellt wurden, wird dieser Abschnitt einen Gateway aufzeigen, der eine Verbindung beider nicht-kompatibler Plug&Participate Welten ermöglicht. Wie aus obigem Abschnitt zu erkennen war, haben Jini und Pini diverse Eigenschaften gemeinsam, insbesondere beruhen sie auf der gleichen Architektur. Andererseits unterscheiden sie sich in Details, so daß sie nicht direkt miteinander kooperieren können. Unter Kooperation wird in diesem Zusammenhang die Möglichkeit verstanden, daß Jini Services auch in der Pini-Welt (natürlich mit eingeschränkter Funktionalität) verfügbar sind und umgekehrt, daß Pini Services auch in der Jini-Welt genutzt werden können.

JP-Gateway im Überblick

Bei Gateways der im folgenden vorzustellenden Technologie handelt es sich sowohl um Jini-Entities als auch um Pini-Entities. Diese speziellen Entities stellen den Dienst bereit, Services der einen Plug&Participate Welt in der jeweils anderen Welt zu offerieren. Dennoch gilt, daß sich der Gateway aus der Sicht der Plug&Participate Technologien wie ein Client verhält; er stellt keinen direkt nutzbaren Service dar. Solche Gateway Clients ermitteln zunächst die Infrastruktur der beiden Plu&Participate Welten, indem sie die entsprechenden Discoveryprotokolle verwenden. Anschließend registrieren sich die Gateways bei den gefundenen Lookup Services, um per Remote Event Feature über alle auftretenden Ereignisse informiert zu werden. Somit besitzt ein Gateway insbesondere immer ein aktuelles Abbild der jeweiligen Gemeinschaften. Sofern der Gateway über entsprechende Ereignisse informiert wird, muß er entsprechend reagieren, z.B. einen neuen Service in der jeweils anderen Plug&Participate Welt offerieren, einen Service aus der Gemeinschaft austragen, veränderte Service Attribute bekannt geben etc.
Weiterhin ist damit auch deutlich zu erkennen, daß Gateways an der Jini-Welt partizipieren. Dies hat zu Folge, daß Gateways auf Jini-fähigen Hosts residieren müssen und vor allem auch die Klassen beider Plug&Participate Technologien vorliegen müssen. Die folgende Abbildung wird die Architektur nochmals verdeutlichen.

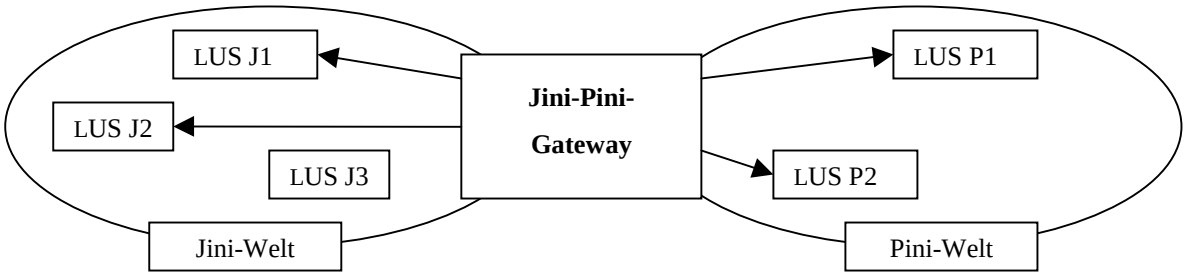


Abbildung 1; Allgemeiner Aufbau des JP-Gateways (LUS = Lookup Service)

Jini _ Pini Gateway

Um allgemein einen Service in einer Pini Gemeinschaft anzumelden, muß der Service Proxy verfügbar sein, aus welchem dann die `ServiceDescription` generiert werden kann. Da der Gateway auf einem Jini-fähigen Host residiert, ist dieses Proxyojekt auch verfügbar, indem es per Remote Class Loading geladen werden kann. Durch den Aufruf der Methode `register(...)` des Pini Lookup Service Proxystubs kann der Jini Service in der Pini-Welt angemeldet werden. Allerdings gilt, daß der Gateway in der Pini Gemeinschaft nun für den eigentlichen Service gehalten wird, obwohl der „echte“ Service in der Jini-Welt existiert. Somit gilt also, daß der Gateway als Pini Service arbeitet, sich andererseits auch als Jini Client verhalten muß. Um ein solches Verhalten zu erreichen, müssen verschiedene Klassen generiert werden:

- *Gateway Intermediate Service*
- PiniRMI Skeleton

Somit ist deutlich zu sehen, daß der Gateway Intermediate Service und der PiniRMI Skeleton den Pini Service verkörpern, welcher den Jini Service in der Pini Gemeinschaft darstellt. Des weiteren arbeitet dieser Gateway Intermediate Service auch als Jini Client, welcher den entsprechenden Jini Service nutzt.
Folgende Abbildung wird das oben beschriebene Prinzip nochmals verdeutlichen:

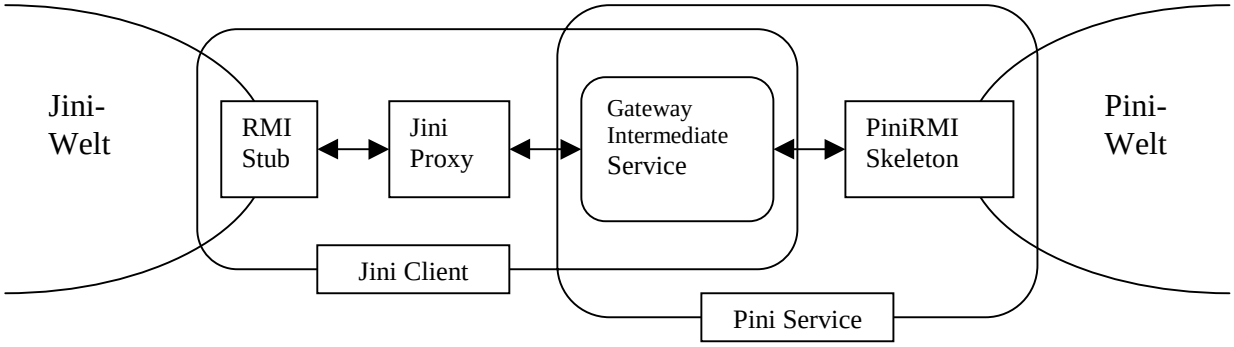


Abbildung 2; Aufbau des Jini _ Pini Gateways

Der beschriebene Gateway Intermediate Service wird dabei vom Gateway erzeugt, wenn der jeweilige Service erkannt wird (also ein entsprechendes Remote Event aufgetreten ist). Der PiniRMI Skeleton wird ebenfalls durch den Gateway unter Zuhilfenahme des im Abschnitt „Jini und Pini – ähnliche, dennoch gegensätzliche Plug&Participate Technologien“ erwähnten Compilers generiert. Der Gateway Intermediate Service wird dann in einem eigenen Thread des Gateways ausgeführt, so daß die Bearbeitung weiterer Services nicht blockiert ist.
Im folgenden wird nun ein Beispiel dargelegt, welches eine Nutzung des Gateways verdeutlicht: Angenommen, es existiert ein **Jini Service** mit folgendem Interface.

```
interface AdditionService{
      public int add(int a, int b);
}
```

Dieser Service erwartet zwei Zahlen, deren Summe an den Client zurückgegeben wird. Der Aufruf der `add(...)`-Methode einer Implementierung des obigen Interfaces verursacht somit einen entfernten Methodenaufruf beim (Jini) Service, welcher die beiden Zahlen addiert und die Summe zurückgibt. Insbesondere folgt nun, daß ein Pini Client zunächst die `add(...)`-Methode seines Proxystubs aufruft. Dieser generiert nun einen RPC-Methodenaufrufrequest und sendet ihn zu dem entsprechenden Skeleton, welcher beim Gateway (Intermediate Service) residiert. Dieser Skeleton richtet den Methodenaufruf an den Gateway Intermediate Service, der als Folge dessen die `add(...)`-Methode des Jini Service Proxys aufruft. Von nun an erfolgt eine einfache Nutzung eines Jini Services, welche hier nicht weiter betrachtet werden soll. Dagegen wird die folgende Abbildung das oben beschriebene Prinzip nochmals verdeutlichen:

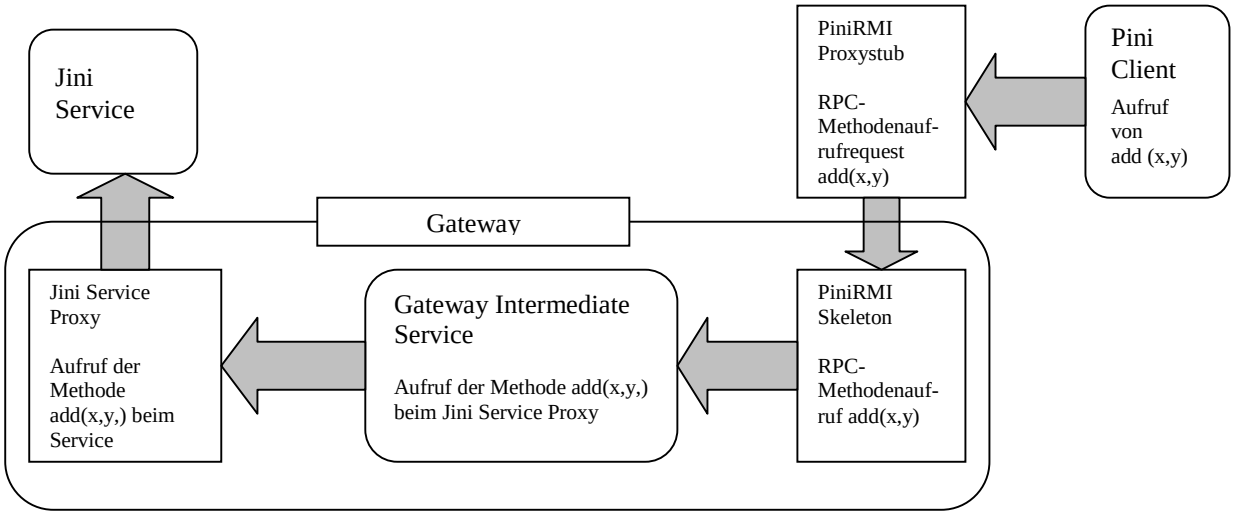


Abbildung 3; Ablauf eines Methodenaufrufes eines Pini Clients bei einem portierten Jini Service

Pini _ Jini Gateway

Das Offerieren eines Pini Services in der Jini-Welt funktioniert im wesentlichen wie der oben beschriebene umgekehrte Fall: Nachdem ein Service erkannt wurde, muß er in der Jini-Welt angemeldet werden. Auch hier müssen die entsprechenden Klassen generiert werden:

- Gateway Intermediate Service
- PiniRMI Proxystub
- Jini Service Proxy
- RMI Stub

Dabei gilt wiederum, daß der Gateway Intermediate Service als Client und als Service arbeitet. Er bietet den Pini Service als Jini Service in der Jini-Welt an und leitet alle Servicemethodenaufrufe an den Pini Service weiter. Folglich arbeitet der Gateway Intermediate Service sowohl als Jini Service als auch als Pini Client. Wie im vorherigen Fall, wird auch in diesem Fall der Gateway Intermediate Service durch den Gateway in dem Moment erzeugt, in dem ein neuer Service in die Pini Gemeinschaft eintritt. Gleiches gilt für den PiniRMI Proxystub, Jini Service Proxy und den RMI Stub.
Somit läßt sich resümieren, daß der Gateway Intermediate Service und der PiniRMI Proxystub als Pini Client arbeiten, wohingegen der Gateway Intermediate Service, der Jini Service Proxy und der RMI Stub den Jini Service darstellen.
Die folgende Abbildung verdeutlicht diese Architektur nochmals:

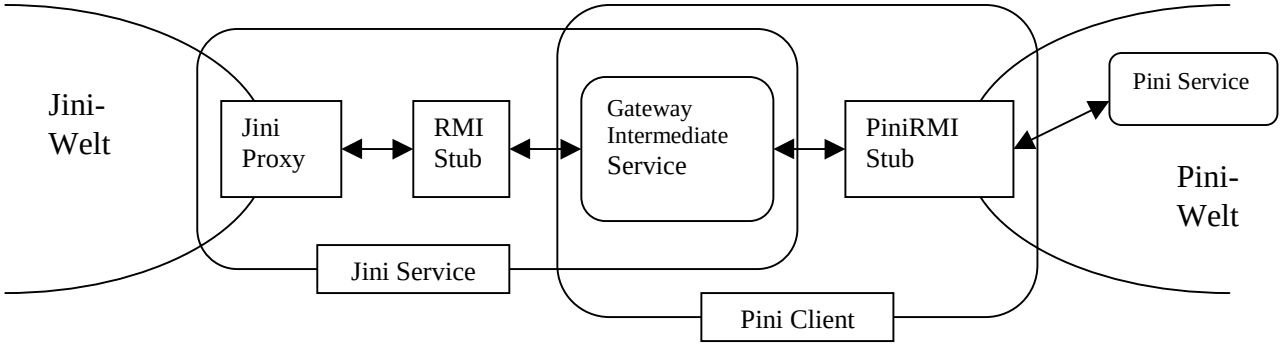


Abbildung 4; Aufbau des Pini _ Jini Gateways

Erfahrungen und Erweiterungen

Erste Anwendungen des JP-Gateways in der Praxis (Laborpraxis) ergaben, daß es in einfacher Weise möglich ist, Pini Services, die auf einer KVM/CLDC Plattform laufen, in einer Jini Gemeinschaft nutzbar zu machen. Zu beachten ist, daß sich die Pini Services lediglich in der Pini Gemeinschaft anmelden mußten. Danach wurden diese vom JP-Gateway erkannt und in der Jini-Welt offeriert. Hier war allerdings eine gewisse Verzögerung zwischen dem Erkennen des Services und der Anmeldung in der Jini Gemeinschaft zu sehen. Diese Verzögerung ist insbesondere auf die Erzeugung der entsprechenden Klassen zur Laufzeit zurückzuführen. Auch die andere Richtung konnte erfolgreich getestet werden, wobei ähnliche Verzögerungen durch das Erzeugen der notwendigen Klassen auftraten. Diese Verzögerungen sind tolerierbar, da sie lediglich einmal bei der Anmeldung des Services in der jeweils anderen Plug&Participate Welt auftreten. Während der Servicenutzung waren keine wesentlichen Verzögerungen zu beobachten.
Weiterhin wurde der Ressourcenverbrauch getestet. Hierbei ergab sich, daß im wesentlichen Ressourcen durch die Bereitstellung von Jini Services/Komponenten verbraucht wurden. Im Gegensatz dazu war der Ressourcenverbrauch durch die Pini Services/Pini Komponenten eher als gering einzuschätzen.
Ferner muß auch das Verhalten der Gateway-Struktur untersucht werden, wenn sehr viele Services durch den Gateway in die jeweils andere Plug&Participate Welt portiert werden. Hier ist vermehrter Ressourcenkonsum durch die Notwendigkeit eines einzelnen Threads für jeden der zu portierenden Services zu beobachten. Aus diesem Grunde muß diese bisher recht einfache Gateway-Struktur um ein Lastbalancierungssystem erweitert werden.
Dieses System sollte derart gestaltet sein, daß die zu bearbeitenden Services gleichmäßig auf die verfügbaren Gateways verteilt werden. „Gleichmäßig“ bezieht sich hierbei allerdings nicht auf die Anzahl der zu bearbeitenden Services pro Gateway, sondern vielmehr auf den wirklich erzeugten Ressourcenkonsum der Services. Die Lastbalancierung sollte möglichst erreichen, daß alle Gateways gleichermaßen belastet werden. Zudem muß

sichergestellt werden, daß ein Service nicht gleichzeitig von mehreren Gateways bearbeitet wird bzw. daß alle Services von genau einem Gateway bearbeitet werden.
Das derzeit implementierte Lastbalancierungssystem der Gateway-Architektur macht sich dabei die Jini Architektur und deren Fähigkeiten zu Nutze. Die Gateways werden um die Eigenschaft erweitert, daß sie Jini Services darstellen und sich daher in der Jini Gemeinschaft anmelden müssen. Das Serviceinterface hat folgenden Aufbau:

```
interface Gateway{
      public void portService(ServiceID id);
      public GatewayState getState(int key);
}
```

Zusätzlich existiert ein Attribute *Master*, welches anzeigt, daß der jeweilige Gateway als Master arbeitet. Dieses Flag wird dann gesetzt, wenn kein anderer Gateway in der Gemeinschaft ist bzw. kein anderer Gateway dieses Flag gesetzt hat. Insbesondere prüfen die Gateways periodisch, ob noch ein Master-Gateway existiert. Weiterhin ist zu beachten, daß nur der Master berechtigt ist, auf das Auftreten neuer Services entsprechend zu reagieren: Er hat die Aufgabe, diese (neuen) Services gleichmäßig auf die verfügbaren Gateways zu verteilen.
Der Master-Gateway hat, wie alle Gateways, stets einen aktuellen Überblick über die gesamte Gemeinschaft, somit kennt er mithin auch alle Gateways. Sofern nun ein neuer Service im Netzwerk auftritt, fordert der Master von allen anderen Gateways den derzeitigen Ressourcenstatus sowie die derzeit vom entsprechenden Gateway bearbeiteten Services mittels der Methode `getState(...)` an. Anhand dieser Informationen wählt der Master-Gateway einen passenden Gateway aus (unter Umständen sogar sich selbst, wenn er derzeit über freie Ressourcen verfügt), der diesen Service zu bearbeiten hat. Diesem ausgewählten Gateway übermittelt der Master die `ServiceID` des zu bearbeitenden Services, so daß der ausgewählte Gateway erkennt, daß er den neuen Service zu verarbeiten hat.

Zusammenfassung

Ausgehend von der Notwendigkeit, Mobilität, flexible Strukturen etc. bereitstellen zu müssen, wurden zwei Plug&Participate Technologien vorgestellt. Dabei zeigte sich, daß beide Technologien auf weitestgehend derselben Architektur aufbauen, dennoch aber für verschiedene Plattformen entwickelt wurden: Jini wurde zwar ursprünglich mit dem Ziel entwickelt, Geräte mit eingeschränkter Ressourcenausstattung in ad-hoc Netzwerke einzugliedern, dennoch ist dieses Ziel bisher nicht erreicht worden. Pini dagegen wurde direkt für den Einsatz auf „limited devices“ designed und konnte auch auf der Plattform KVM/CLDC erfolgreich getestet werden. Somit zeigte sich im Vergleich beider Technologien, daß sehr große Gemeinsamkeiten bestehen, aber dennoch beide Technologien nicht direkt miteinander kooperieren können. Aus diesem Grund wurde die hier in diesem Artikel vorgestellte JP-Gateway Technologie entwickelt, welche es nun ermöglicht, Jini Services auch für „limited devices“ nutzbar zu machen bzw. Services, die von „limited devices“ bereitgestellt werden, auch in der Jini Welt anzubieten. Insbesondere konnte gezeigt werden, daß diese Portierung in die jeweils andere Plug&Participate Welt passiv geschieht. Dies bedeutet, daß ein Service nicht aktiv dafür verantwortlich ist, auch in der jeweils anderen Plug&Participate Welt verfügbar zu sein. Es reicht aus, sich entweder in der Jini Welt oder in der Pini Welt anzumelden.
Somit gilt für Clients, daß sie durch das Lookup Protokoll Informationen über alle verfügbaren Services aus beiden Welten erhalten. Im Vergleich dazu muß ein Service, welcher mittels der *Jini Surrogate Architektur* der Jini Welt beitritt, aktiv bemüht sein, einen *Surrogate Host* zu finden, um dann aktiv der Jini-Welt beizutreten (für weitere Details zur Surrogate Architektur siehe [8])

Literaturquellen

[1] K. Arnold, B. O’Sullivan, R. W. Scheifler, J. Waldo, A. Wollrath; 1999:
The Jini™ Specification; Addison-Wesley;
ISBN: 0-201-61634-3

[2] W. K. Edwards; 1999:
Core Jini™ 2nd; Prentice Hall;
ISBN: 0130894087

[3] J. Gosling, B. Joy, G. Steele, G. Brancha; 2000:
The Java™ Language Specification; Addison Wesley
ISBN: 0201310082

[4] Sun Microsystems:
Java™ Remote Method Invocation (RMI) Specification
http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[5] S. Deter, K. Sohr; 2001:
Pini - A Jini-like Plug&Play Technology for the KVM/CLDC
in: "Innovative Internet Computing Systems"; Springer-Verlag
ISBN: 3-540-42275-7

[6] PABADIS Consortium:
http://www.pabadis.org

[7] Sun Microsystems:
Connected Limited Device Configuration, Version 1.0
http://www.sun.com/software/communitysource/j2me/cldc/

[8] Sun Mircosystems:
Jini™ Technology Surrogate Architecture Specification (Version 1.0)
http://developer.jini.org/exchange/projects/surrogate/