Hierarchical Scheduling for State-based Services

Jens Bruhn, Sven Kaffille*, Guido Wirtz
Distributed and Mobile Systems Group
Otto Friedrich Universität Bamberg
Feldkirchenstrasse 21, 96052 Bamberg, GERMANY
email: {jens.bruhn|sven.kaffille|guido.wirtz}@wiai.uni-bamberg.de

***Abstract*— Service descriptions based on type hierachies and abstract service states ruling the availability of operations permit more secure service combinations in distributed systems design than traditional signatures. The advantages of these additional information about services can also be utilized to make service scheduling more robust and efficient. A framework for scheduling such services is introduced and the central techniques used to provide a portable `Java`-based scheduling framework are discussed.**

Keywords
state-based modelling, distributed services, scheduling

I. Introduction

Developing distributed service structures is a complex task and requires a well-founded methodology ruling the design and implementation process. Service descriptions based on hierarchical type structures and abstract service states permit more secure service combinations in this context than traditional interfaces based more or less on syntactical signatures. The *OCoN* approach (*Object Coordination Nets*), developed over the last seven years [WGG97], combines standard OOA/OOD design techniques as, for example, provided by the *Unified Modeling Language* (`UML`) [Obj03] with concepts like interface hierarchies to publish services, state-based resource modeling and a complete language to describe, (partly) analyze and test the correctness of distributed systems via simulation on the design level. For an overview of these aspects of the `OCoN` approach, refer to [GW01].

In specifying the point where service users and service providers interact, *service interfaces* are in the very center of the approach. *OCoN interfaces* are an extended version of standard interfaces as used in, e.g., `Java` [Gos00], `CORBA` [Obj02] or the `UML` [Obj03]. Additionally to the well-known signature, they use so-called *protocol net*s (`PN`), i.e., *state machine*–like Petri-Nets to reflect possible state changes of service providers iff they cannot be hidden from service users by an implementation. The states (places) of a PN abstract from all internal states of a service provider to those states that may be of interest for the service user (see section II for an example). If, for example, a specific interface operation is only available in a specific state, it is important for the user to know about that fact, because a call to such an operation may block the caller until the service enters a state where the operation becomes available again. Based on such information, a user may decide to wait, to search for a service provider in a more convenient state or to abandon calling the service. Hence, state-based interfaces are a really useful concept to describe services and to check whether a service provider uses implicit assumptions or non-robust implementations which may harm a calling user.

On the other hand, enhanced service descriptions are also useful on the provider side because they offer additional abstract knowledge about the interdependencies of services, their effects on provider states and so on. In the case of a *service broker* that offers lots of different services this knowledge may be used to enhance scheduling decisions. This paper describes a framework that provides the mechanisms for scheduling with respect to service types, attributes and service states. The framework is `Java`-based and uses `Jini` technology [SM03a].

Matching of requests to available services in the scheduling facility is done by using so-called *interface templates* for service descriptions. The mechanism supports interface hierarchies in a manner that user requests for an interface may be served by an exactly matching or a more specific interface still matching the request's template. Moreover, requests to interfaces in specific states are handled as well as state-specific priority strategies in order to fulfill as much requests as possible.

The rest of the paper is organized as follows. Section II presents a simple example for a state-based service that explains the `OCoN protocol net` concepts and is used later on as a running example. Section III describes the architecture of the scheduling system, section IV discusses the main implementation concepts used and section V sketches the usage of the entire system. The paper closes with some remarks on future work.

II. Example Service Interface

As an example, we use a `WarehouseSection` which represents an area of a warehouse inside a warehouse-management-system. The section can be dynamically reserved for a particular type of item. Figure 1 (see next page) shows the correspondending protocol net for this interface. There are three possible states (represented by hexagons) for a `WarehouseSection`:

1) The state `[empty]` indicates that there are no parts available inside the section.

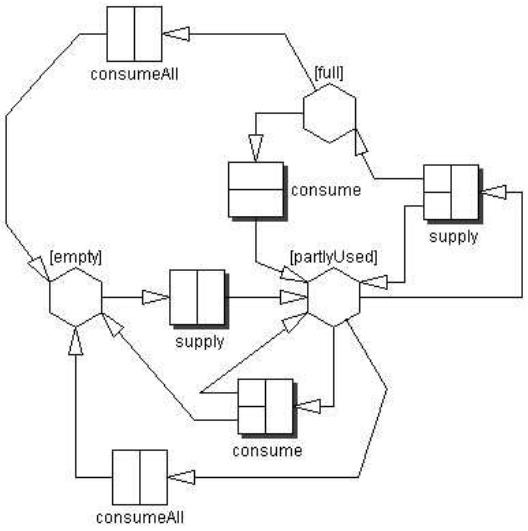

Fig. 1. `ResetableWareHouseSection` Protocol Net

2) The state `[partlyUsed]` indicates that there are parts available inside the section but that there is still free space for additional ones.
3) In the state `[full]` all space of the section is occupied by items.

Note that the states presented above apply to warehouse sections with capacities greater or equal to two items of the particular type. For the rest of this paper we assume this to be given.

A `WarehouseSection` supports two service operations:
`+void supply(item i)` adds an item to the section. The method is only usable if there is still space available, i.e. in the states `[empty]` and `[partlyUsed]`.
`+Item consume()` removes an item from the section. Note, that the section has to be either in the state `[partlyUsed]` or `[full]` for applying this method.

State-sensitive methods are indicated by arcs starting at states and ending at the transition boxes, e.g., a `consume`–transition is reachable from the hexagons representing `[partlyUsed]` and `[full]`. Alternative (non-deterministic from the users perspective) resulting states of a particular method are visualized by transitions with alternative output arcs.

Consequently, the different operations are not available from all of the possible states. Moreover, the type of the correspondent items represents a marking for the particular `WarehouseSection` which indicates that the section service is only usable in combination with special items. For the users of such a service – e.g. a supplier which delivers items or a warehouse worker that fetches items – the instance of a section is not of interest. They just seek a section which is marked according to their needs and is in one of the states from which the desired operation is executable.

A `ResetableWarehouseSection` is an extension of a `WarehouseSection` by adding the new method
`+ item[] consumeAll()`.

Using this method, a consumer may retrieve all available parts from the `WarehouseSection`. This method is available in the states `[partlyUsed]` and `[full]` and leads always to the state `[empty]`.

III. Architecture

The architecture of the framework distinguishes three different perspectives, the so called *views* (see figure 2). The *Service view* includes all parts of the framework which are necessary for the interaction of a service with internal components and clients. The *Client view* represents the correspondent parts of the framework for the client side. Within the *Internal view* all components for the execution of the infrastructure of the system and the interaction with services and clients reside.

Orthogonal to the views, the framework architecture is organized into three layers. At the bottom of the layer hierarchy resides the *Communiction layer* which is responsible for the delivery of messages between the different participants of the system. It also provides a mechanism to discover an entry point via a so called *Lookup*. This layer provides interfaces which abstract from the concrete middleware to the higher layers. This kind of encapsulation supports the migration to other communication infrastructures without any changes on higher framework layers.

Above the Communication layer the *Framework layer* is located. It contains all parts of the system which are not directly touched by developers of services or clients. It provides the *Starter*, *Creator* and *Scheduler* components which are started within a network to take over different administration tasks and the scheduling itself.

The starters act as platforms upon which the other components can be dynamically started during runtime. The creators are responsible for the administration of the system. At any time there is only one specific creator allowed to administrate the system. It triggers the instantiation and/or deactivation of creators and schedulers upon starters. All other creators act as replicas to keep track of the system state and to take over administration in case of a crash or shutdown of the main creator.

Schedulers are the core components of the framework which deliver the scheduling facility for services and clients. They are arranged hierarchically according to the hierarchy of the scheduled interfaces and are – similar to the creators – replicated. There is always one scheduler responsible for a particular interface and the correspondent services and clients.

On top of the layer hierarchy resides the *Application layer* which includes interfaces for the interaction with the system for developers of services and clients. It masquerades all steps of interaction with the components of the framework and permits to interact directly with the desired services and clients. The communication between components can be distinguished into three groups. The first group includes every interaction which deals with the *discovery* of other components of a running system

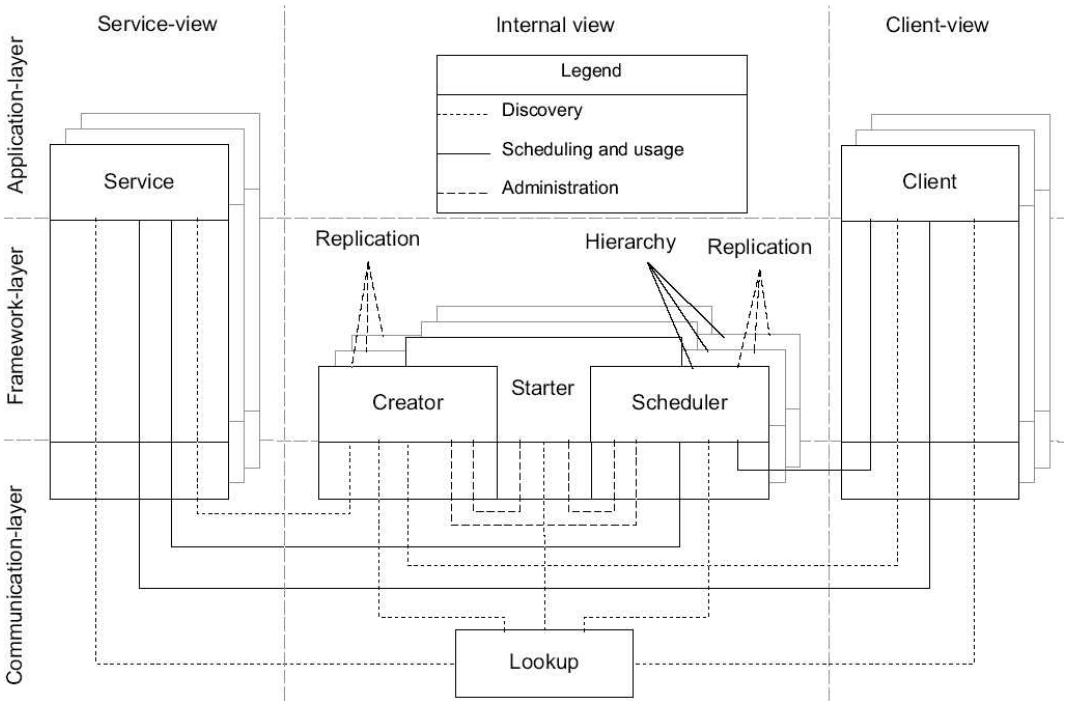

Fig. 2. System architecture

(dotted connections in figure 2). Every component which wants to find other ones uses the lookup to discover the main creator. The responsible creator keeps track of all components of the internal view in the network and can be asked for references to them.

Every interaction which deals directly with the scheduling and usage of services is summarized in the second group which includes the communication between services, clients and schedulers as well as among the scheduler hierarchy (continuous lines in figure 2). The third group of interactions summarizes communication for *internal administration* purposes (dashed lines in figure 2). It subsumes the communication between creators and starters, creators and schedulers and vice versa.

IV. Concepts

This section discusses the concepts used to organize and implement the scheduling framework.

Initialization and System Administration

The first components of a scheduling system which are initialized are the *lookup* and at least one *starter* upon which the main *creator* is started. If further starters are created they use the lookup to find the main creator with which they register. The main creator keeps track of all starters in the system and uses them to initiate the instantiation of new creators and schedulers. Schedulers for new interfaces and the correspondent hierarchy are created on requests of services only (see below). For creators and schedulers there exists a lower bound which indicates how many replications of each creator and scheduler are desired within the system. In regular intervals, the state of the system is checked by the responsible creator in five steps. First the creator analyzes if there is another responsible creator available in the system. If this is the case the detecting creator decides, based on the identification numbers, which creator has to be deactivated. The shutdown of this creator is then initiated immediately. The situation of two responsible creators coexisting in a system is only possible in exceptional cases e.g. when two system partitions are brought together. In a second step the creator connects to all starters, creators and schedulers to check if they are still available and no crashes have occured since the previous check. If a component is not available the reference is removed from the creator. If the number of still available creators falls below the lower bound, the creator selects a number of starters upon which new replica should be instantiated. Each starter which has not reached it's limit of componenents to start and which has not yet started a creator is a possible candidate for the instantiation. Upon these candidates new creators are started up to an upper bound of replicas – which represents the maximal desired number of creators in the system – or the point when there are no candidates left. The third step includes the same procedure for the schedulers. During this proceeding it is ensured that there does exist only one scheduler for a particular interface upon each starter. The fourth step deals with the possibility of two or more schedulers being responsible for a particular interface. If such a situation is identified, all except one responsible scheduler are deactivated. Similar to the possible existence of two responsible creators this situation may only occur

in exceptional cases. The last step includes the search for and the deactivation of idle schedulers. A scheduler is idle if there are no services of lower schedulers associated with it any more. In this situation the scheduler is not able to assign client requests to services and consequently is of no use for the system. First, candidates for deactivation are identified and, after a certain amount of time to give them a second chance, rechecked. If they are still ilde, they are – including all replica – shut down.
Informations about not yet available or new components are replicated to all creators at once. All replicated creators try to reach the main creator at regular intervals. If they cannot reach it they elect a new leader among themselves. This proceeding guarantees a high degree of reliability and the possibility of loss of information is minimized.
Moreover, possible conflicts regarding two or more responsible creators and/or schedulers coexisting in one system are solved and no intervention by an administrator is necessary. There are only three situations conceivable when this is required:

- If all lookup facilities in the system have crashed, at least one new instance has to be started.
- If all creators (responsible and replicas) have crashed, a new creator must be started.
- If there is no capacity left upon starters and new schedulers should be instantiated, an administrator has to extend the capacities of the existing starters or launch new starters into the system.

This leads to a high degree of fault tolerance in combination with a minimized need for manual administration during runtime of a system.

Basic State-based Scheduling
Protocol nets are mapped to (*Java*) interfaces that are implemented by services. All services implementing the same protocol net are registered with the same scheduler. In our warehouse example all warehouse sections (instances of type `WarehouseSection`) register with one and the same scheduler. When a warehouse section (service) starts up it gets – via the lookup – a reference to the main creator from which it requests a reference to the scheduler responsible for the `WarehouseSection` interface the service implements. If no such scheduler exists the creator starts one at a node in the network and a reference to this newly created scheduler is given to the service. Schedulers for interfaces (if any) that are super types of the interface for that the scheduler is currently requested are also created if not yet existing and the scheduler hierarchy is built up. Schedulers for sub interfaces are created later if services for sub interfaces arrive and are also attached to the scheduler hierarchy. By this proceeding schedulers are instantiated on demand and schedulers for new interfaces can be integrated easily into runnning systems. The service registers with the scheduler and from that time the service has a so called service proxy associated with it to enable the scheduler to communicate with the service and the other way round. A client that wants to use a `WarehouseSection` searches for a creator to obtain a reference to the scheduler responsible for the interface and – if available – registers with that scheduler. Different to the proceeding for services no new instances of schedulers are created upon requests of clients. This is not done, because only if services are available for a particular interface, client requests can possibly be fulfilled. If a service exists this service will initate the instantiation of a correspondent scheduler. If a scheduler exists, also the client is associated with a so called client proxy that is used by the client to request a service and by the scheduler to notify the client when its requests can be fulfilled. In order to be able to use a service the client offers a request to the scheduler. There are two possibilities for a client to request a service:

- request a particular method from a service type. In our example perhaps a client wants to get an item from a `WarehouseSection` by the method `consume`.
- or request a service in a particular state. In our example the client may ask for the state `[partlyUsed]` to consume an item.

If one instance of `WarehouseSection` is in the state that matches the client's request the service is marked as busy. Then the client is notified via its client proxy that a service is available. The client is assigned to the service instance by the scheduler via its service proxy that creates a ticket which is given to the client and to the service. The client gets a reference to the service and the ticket so that he can invoke the desired `consume` method. If it has requested the service in a particular state, all methods permitted in that state may be invoked and after that all methods permitted from the service's resulting state can also be executed by the client and so on. Consequently by requesting a service in a particular state (e.g. in our example `[partlyUsed]`) it is possible to make subsequent invocations according to the service's protocol net on that instance depending on the resulting state. This requires the client to have knowledge about the service's protocol net and the client has to test for the service's state before it invokes the next method. Note that our framework prevents invocations to methods that are not permitted to be executed from a service's current state by throwing an exception to the client that tries to make an illegal invocation.
The ticket is necessary to prevent clients from invoking methods of services without requesting them via the services' scheduler. To prevent the service from starving if the client does not use the ticket (e.g. it crashes) the service waits for a certain time. After that time he notifies its service proxy and the request is given back to the scheduler and put back into the scheduler's queue. If this happens more than once to a request, the request is deleted from the queue and the client is notified about it if still alive. In the case (as in our example) that a method

has been requested and the method has been executed the proceeding is as follows. After the client has finished its invocation of `consume` the service involved invalidates the ticket, propagates its new state (maybe `[empty]` or `[partlyUsed]`) to the scheduler via its service proxy and asks for a new request to fulfill. The latter can be omitted if the service desires to stay idle or wants to shut down. If a service has been requested in a particular state there is also a certain time between subsequent method invocations during that a client has to make its next invocation. After this time has elapsed once, the service again asks for a new request to fulfill. This also can be omitted if the service desires to stay idle or wants to shut down.
The requests of clients are hold in a queue at the scheduler and the first request that can be fulfilled by a service instance is assigned to that service. As the scheduler is responsible for all services of a particular type (here `WarehouseSection`), a client does not request a specific instance of a service but only the service type.
To facilitate requests for a service with more specific properties, services are associated with so called *service templates*. A template describes a service's properties and is held at the scheduler within the service proxy. If a service changes its template this has to be propagated to the service proxy immediately. Clients can request services by specifying a template describing the desired properties of the requested service. These templates are compared to each other by the scheduler to assign a service to a client. In our example, a `WarehouseSection`'s template may contain the type of the stored items in order to make it possible for clients to consume or supply an item of a specific type. If a client does not specify a template all services of a requested type can be assigned to the client if in a matching state. If there is no service available matching the template of the client's request the client is notified. This can happen immediately after the request has been made or if the client's request has been in the scheduler's queue for a while and all services that could have fulfilled the request as of their templates have been shutdown. The scheduler is able to make simple computations to detect clients' requests that can successively be executed by a service instance and assign such requests directly to this service instance. The service and client proxies are responsible for the detection of service/client crashes. If a proxy detects that its associated service/client is not reachable it logs off from the scheduler to enable the scheduler to clean up.

Interface and Scheduler Hierarchies
Often, services are organized in a inheritance hierarchy between some interfaces (resp. protocol nets). So the most specific interfaces are also of the type of their direct and indirect super types. For this reason a service may also fulfill requests made for its direct and indirect super types. Therefore schedulers for services that implement interfaces of a particular hierarchy are organized hierarchically, too. The hierarchy of schedulers is maintained by the creator. To implement this hierarchy, the scheduler of an interface at a lower level is registered like a service with the scheduler of the direct super type of its interface. This makes it transparent to the parent scheduler with regard to the assignment of requests if there is any lower scheduler. The lower scheduler can ask for requests at its parent scheduler, e.g., when its services are idle and/or at regular intervals. A scheduler is therefore also associated with a proxy at a parent scheduler. This proxy acts (from the view of the parent scheduler) like a service's proxy. Only the process of registration of a scheduler with a parent scheduler is different from the registration of services. Currently the framework only supports the registration of a scheduler with one parent scheduler. Therefore currently no multiple inheritance for protocol nets is supported.
In our example, a `ResetableWarehousSection` offering an additional `consumeAll` operation may also fulfill requests from clients that are not aware of this feature. If there is no service of type `WarehouseSection` that is able to fulfill the request of a client to `supply` an item, this request can be fulfilled by an implementation of the interface `ResetableWarehouseSection` if the template the client desires matches the template of any `ResetableWarehouseSection`.

Scheduler Replication
As the crash of a scheduler would cause the failure of the communication between services and clients, the schedulers are replicated. Replicas are instantiated by the creator and register with the currently responsible scheduler. In order to provide information for the replicas to identify the proxies of the clients and services registered, proxies have unique identifiers. The replicas are notified of the following events, and the data in brackets are transmitted to them:

1) Registration of a client (client's proxy identifier, reference to client) to enable a scheduler's replica to contact the client in the case it has to take over control.
2) Registration of a service (service's proxy identifier, reference to service), to enable a scheduler's replica to contact the service in the case it has to take over control.
3) Arrival, deletion and fulfillment of a client's request (request) to preserve the order of requests clients have made.

More information is not necessary (e.g., services' templates) as they are collected from the involved services and clients at the time a replica of a scheduler becomes the responsible scheduler. If one component in the system (e.g., a service) recognizes that the responsible scheduler for an interface has crashed it tries to obtain a new reference from the creator. The creator declares a new responsible

scheduler and gives the reference to the component that has detected the crash. That component announces the identifier of its proxy to the new responsible scheduler to reestablish the association to its proxy. The new responsible scheduler contacts the clients and services involved via their proxies so that they can update their references to the responsible scheduler and the scheduler can match its information about its state with the information of the clients and services. For example the scheduler asks every client it knows for its requests and if a client's information about requests differs from the scheduler's the scheduler adapts the new information. In this way the scheduler can come back to a consistent state. If in the meantime a service or client recognizes the crash, it acts like the component that first detected the crash but is blocked until the new responsible scheduler has taken over control. After that it gets a reference to the new responsible scheduler. If the scheduler has reestablished all connections to the services and clients it continues work. Reestablishing the connections to clients and services comprises removal from the scheduler of clients' and services' proxies and clients' requests to which the corresponding connections could not be reestablished. This proceeding ensures that no information about a scheduler's state is lost or a scheduler has information that is no longer valid. If, for example, a replica that becomes responsible did not receive information about a service that has connected with the scheduler, that was responsible before, these information is not lost, as the service itself reregisters with the scheduler at the time it detects that the formerly known one has crashed. A scheduler's connections in a scheduler hierarchy in case of a crash are reestablished with help of the creator. For connections to lower schedulers the corresponding proxies are responsible if any registered. A scheduler's proxy maintains the connection to its scheduler and, if it recognizes that the connection is broken down, reestablishes the connection with help of the creator by requesting the scheduler like a client does. If it gets no new reference to its scheduler the proxy logs off from the scheduler it is registered at. The communication layer component of a scheduler is responsible for connecting to a parent scheduler. If it recognizes that the connection is broken, it requests a reference to its parent scheduler from the creator like a service does.

V. Usage

The architecture and concepts described in this paper have been implemented in a `Java`-based (version 1.3+)[Gos00] framework by applying the Jini network technology (version 2.0)[SM03a][SM03b]. Jini can be replaced by any other middleware by replacing parts of the framework's communication layer. These parts are separated from the communication layer's middleware independent components. The communication layer's middleware independent components are for example responsible for replication of schedulers.

Replication of schedulers and other framework internal components are hidden from the clients and services as the connections to these components are managed by the framework's communication layer. If, for example, a scheduler crashes the communication layer's middleware independent components detect that and all calls to the scheduler (from clients or schedulers) are transparently blocked until a new scheduler has taken over control. Clients and services do not recognize such a situation except that calls to the scheduler may take longer while the framework recovers from the crash.
The complete framework is published under the GNU General Public License and can be obtained from [Dis04].

Service implementation
To implement a service a class named `OCoNService` has to be extended. By extending this class the connection to the framework is made available when the service is created. In order to deploy a service, a definition of the protocol net the service implements must be provided in an XML file. This file is read by the service when it is created. The service automatically registers with the scheduler responsible for its type. The class `OCoNService` also provides methods to manipulate the service's template. To provide the desired functionality the class extending `OCoNService` has to implement the interface that implements the protocol net for the service. For our example possible interfaces are `WarehouseSection` or `ResetableWarehouseSection`. These interfaces must extend the framework interface `ServiceInterface`. The XML file mentioned above provides the connection between the service's state and its methods. This information is used by the framework to ensure that no invocation to a service's method is made while the service is not in a state required for execution of that method. The implementation of a service in every method has to set the state the service is in after execution of the method to enable the framework to correctly assign clients.

Client implementation
Clients can connect to the framework by instantiating the class `RequestManager`. From that class clients can obtain references to services (proxies) on which services' methods can be invoked as if the services were local objects. The proxies transparently make requests for the corresponding service type, so that a client does not have to care about making requests. Requests can be associated with a template so that the invocations made on a proxy are only made to services that match the given template. The desired template has to be given to the `RequestManager` instance at the time a reference to a service is requested from it. It simply calls a method on a local object. The proxies can be *synchronous or asynchronous*. Synchronous Proxies block a client until the request is fulfilled or it becomes impossible to fulfill (e. g. the last service matching the template of the client's request shuts

down). Asynchronous proxies return immediately from the called method and the result of the invocation can later be obtained from the `RequestManager`. Clients additionaly must implement an interface that provides a single method that is used by a client's `RequestManager` instance to notify it that the connection to the framework has broken down.

Running the system

In order to use the framework at least one Jini lookup service has to be running and reachable. It is used by all components of the framework to find the initial reference to the responsible creator. To prepare the infrastructure for the usage by services and clients at least one starter and one creator have to be instantiated. A starter is created by instantiation of the class `OCoNStarterCompute`. During startup a reference to a configuration file must be handed to the new starter. In a second step the starter must be instructed to instantiate a new creator. This will declare itself as *leader* after a while. Afterwards, the infrastructure is managed by the leader up to the point of it's deactivation or failure. For replication and load balancing purposes, additional starters may be instantiated which will find the responsible creator and register with it to act as platform for new instances of creators and/or schedulers. New starters can be added at any time because they will continuously retry to find the responsible creator if they have no reference to it or the referenced creator is not available anymore.

VI. Conclusion and future work

With the framework presented, the basic functionality for the scheduling of state-based services within a network is provided and may be used by extending the pre-defined `Java` interface and class plugins of the framework.
Due to the abstraction from service instances through the schedulers and the concept of scheduler hierarchy load balancing can be achieved easily. The abstraction with help of the scheduler hierarchy makes it also possible that a request for a service on a higher hierarchy level can transparently be served even if no service of the requested type, but subtypes of this service are available. Clients can only obtain references to services that are in a correct state regarding their requests. With these references they can only invoke those methods they have requested. So there is less possibility for a service to be crashed by a faulty use of a client. Even in the case a client requests a service in a particular state and by that can nearly freely access the service by subsequent method invocations our framework ensures that only invocations permitted in a service's current state can be performed.
If a client only requests methods it is not required that the client has knowledge about the service's protocol net.

Future development of the system is planned w.r.t two directions. From the user's point of view, adapting the framework through a more flexible configuration is desirable. The system will include the ability to exchange the scheduling algorithm to give developers the freedom to adjust the behavior of the system according to their particular needs. It is also considered to provide a facility for on-demand-creation of service instances upon the starters which are not a priori bound to special environments and to keep the state of a created service. This permits the automated migration of services within the network, leads to more flexible load balancing schemes and opens the possibility to deal with shutdown of nodes or to migrate from insecure nodes because of failures within the network. The other main direction deals with the monitoring of the system-state and it's manipulation. For administrators of a system there will be the possiblity to gain information about the allocation of internal components as well as of the workload of particular nodes. It will also be possible to initiate the creation and deactivation of components. This will lead to a much better on-the-fly control of the running system.

References

[Dis04] Distributed and Mobile Systems Group, University of Bamberg: *OCoN-Framework version 1.0*. http://www.lspi.wiai.uni-bamberg.de/dmsg/software/, 2004.

[Gos00] Gosling, James et al.: *The Java Language Specification*. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, 2000.

[GW01] Giese, Holger and Guido Wirtz: *The OCoN Approach for Object-Oriented Distributed Software Systems Modeling*. Computer Systems Science & Engineering, 16(3):157–172, May 2001.

[Obj02] Object Management Group: *OMG CORBA 3.0*, January 2002. OMG doc formal/02-12-02.

[Obj03] Object Management Group: *OMG UML 1.5*, January 2003. OMG doc formal/2003-03-01.

[SM03a] Sun Microsystems, Inc.: *Jini Architecture Specification*. http://wwws.sun.com/software/jini/specs/jini2_0.pdf, 2003.

[SM03b] Sun Microsystems, Inc.: *Jini(TM) Technology Starter Kit*. http://wwws.sun.com/software/jini/specs/newapi2_0.pdf, 2003.

[WGG97] Wirtz, Guido, Jörg Graf and Holger Giese: *Ruling the Behavior of Distributed Software Components*. In Arabnia, H. R. (editor): *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada*, July 1997.