Efficiently and Transparently Automating Scalable On-demand Activation and Deactivation of Services with the Activator Pattern

Michael Stal
michael.stal@siemens.com
Siemens Corporate Technology, Munich, Germany

Douglas C. Schmidt and William R. Otte
{d.schmidt,w.otte}@vanderbilt.edu)
Vanderbilt University, Nashville, TN, USA

Abstract

Computing systems are increasingly designed as a collection of interacting services, which constitute a set of functionality offered by a service provider or server to its clients. Many service-oriented computing systems have constraints on the resources they allocate and manage. In these systems, certain types of services should consume resources only when they are accessed by clients and clients should be shielded from where services are located, how they are deployed, and how their lifecycle is managed. The Activator pattern provides an effective means to efficiently and transparently automate scalable on-demand activation and deactivation of services accessed by many clients. This paper motivates the need for the Activator pattern, describes the structure and dynamics of canonical implementations of the pattern, and examines the benefits and liabilities of applying this pattern to services in resource-constrained computing systems.

1 Introduction

Despite the decreasing costs and increasing size of main memory in computing hardware, many service-oriented systems have constraints on the resources they can allocate and manage. For example, the memory constraints in battery-powered devices, such as devices using Google Android or Apple iOS, require careful programming techniques to ensure applications do not consume excessive power [Power07] or memory resources, which are limited since Android and iOS do not support paging virtual memory to disk to avoid degrading battery power and the effective life of Flash-based storage. [AppleDev] Likewise, distributed traffic control systems and manufacturing plants in the industry automation domain are increasingly implemented using embedded devices known as *controllers* that communicate via high-speed bus interconnects and networks [POSA4]. When software developers build memory-, CPU-, and power-constrained systems, therefore, they must determine how to provide services, such as communication daemons, inventory trackers, system monitors, and command and control services, in a manner that scales gracefully as the number of clients increases.

In service-oriented automation systems, for example, service processing must be scalable when multiple clients access embedded devices simultaneously [EmSOA]. One service deployment strategy is to apply *eager resource allocation* [POSA3], which activates processes in controllers during system initialization and runs all services in processes while the system is operational, irrespective of which services are actually accessed by clients. Embedded devices, however, often have a limited amount of resources, such as main memory, CPU time, network connections, and battery power [SmallMemory]. As the number of clients or services increases, therefore, an eager resource allocation strategy scales poorly because unused server processes consume resources that could be allocated more effectively to services actually being accessed by clients.

A typical scenario in the lifetime of an eager resource allocation strategy for a controller in an industrial automation system is shown in Figure 1. The “System Load” rectangle in the diagram depicts the current memory load of the controller that falls into the range between 0% and 100%. In this eager resource allocation strategy, all services are activated automatically at system initialization and consume significant amounts of available system resources as indicated by increased memory load. In the depicted time span shown in this figure, only Service 1 is accessed successfully by a client increasing memory load to 70% since all other services are in memory waiting for incoming requests. The consumption of resources by allocated—but unused—server processes can therefore increase unnecessarily, including:

- **Service response time**, e.g., by competing for resources with services actually accessed by clients, and
- **Hardware costs**, e.g., by requiring more main memory, CPU, and power consumption than would otherwise be needed to handle clients simultaneously.

Figure 1 shows how a second client tries to access Service 3 but obtains an overload error since the embedded controller has dedicated its memory resources to Service 1 and to the eager allocation strategies of other services. This overload error gets generated because the memory load reaches a predefined overload barrier of 98%. In this example, the controller cannot initiate any new services when overload barriers are reached.

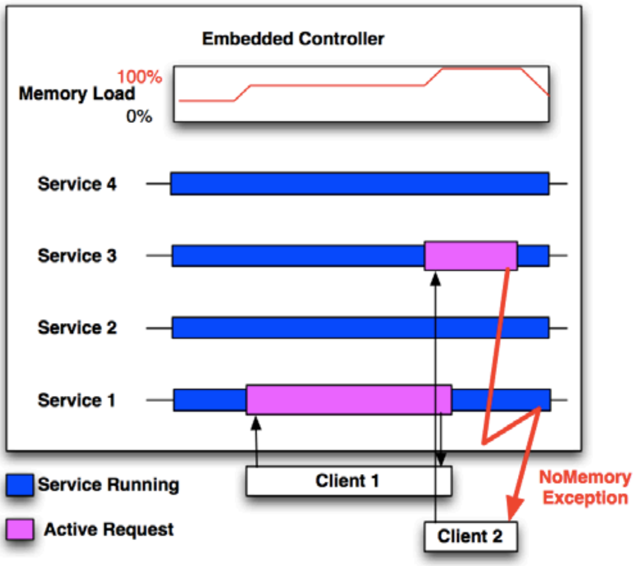


Figure 1: An Eager Resource Allocation Strategy.

More efficient service activation strategies than eager resource allocation are therefore necessary to optimize resource usage and enhance scalability when resources are scarce. Depending on the software technologies used in the automation system, these activation strategies can be implemented using operating system (OS) and middleware superservers, such as Inetd [SteRago08], the CORBA Implementation Repository [HV99], or system-specific variants of these technologies, based on the Activator pattern described in this paper.

The remainder of this paper is organized as follows: Section 2 describes the structure and functionality of the Activator pattern; Section 3 summarizes the activities associated with designing and implementing service-oriented systems using this pattern; Section 4 presents the results of a case study that applied the Activator pattern to an industrial automation system and generalizes the benefits and liabilities of applying this pattern to the case study; Section 5 surveys known uses of the pattern and compares the Activator pattern to related patterns; and Section 6 presents concluding remarks.

2 Structure and Functionality of the Activator Pattern

This section describes the structure and functionality of the Activator pattern.

2.1 Applicability and Problem Domain Scope

The Activator pattern is intended for resource-constrained service-oriented system environments *without* stringent real-time requirements whose services (1) can be accessed by multiple clients simultaneously, (2) require non-trivial utilization of constrained resources, such as memory, power, or processing time, (3) are activated quickly relative to service processing time, and (4) are not accessed continuously throughout the system lifetime. In these types of systems, multiple clients often simultaneously access services (such as e-commerce web services, audio/video streaming services, or system-level OS/network services like DNS, FTP, or SSH daemons) that perform functionality on behalf of the clients. These services are deployed in service execution contexts (such as operating system processes, threads, and/or component containers) and consume scarce system resources (such as network/database connections, threads, virtual memory, process table slots, and open files). In this context, it is often necessary to balance the following *forces*:

- *Parsimony*. Service execution contexts available in the system should only consume resources for services that are accessed actively by clients.

- *Transparency*. Clients should be shielded as much as possible from where services are located, how they are deployed onto hosts in a network, and how their lifecycle is managed.

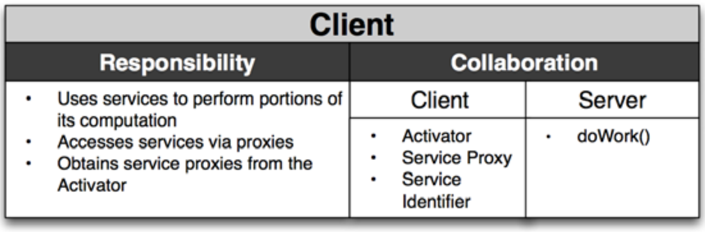
2.2 Solution Approach

To resolve the forces described above, minimize resource consumption by activating service execution contexts on demand, running service implementations in these contexts, and deactivating services and their contexts when they are no longer being accessed by clients. Use proxies to transparently decouple client access from service behavior and lifecycle management. Implement *services* that have *service identifiers* and offer functionality to *client* applications via their *service proxies*. Use *service execution contexts* to manage the lifecycle of these services, in particular their activation, processing, and deactivation. Implement an *activator* that uses an *activation table* to activate service execution contexts on demand and deactivate them when clients no longer access them. Provide a registration interface that services can use to register and unregister their availability with the activator. Use the service proxy to ensure clients only access services via activators. If a service is not running when a client tries to access it, an activator automatically creates the appropriate service execution context and arranges for the service to process the client's request(s) in this context.

The remainder of this section uses enhanced Class Responsibility Collaborator (CRC) diagrams [Fayad03] [Fayad04] and Unified Modeling Language (UML) structure and sequence diagrams [Fowler03] to convey the structure and dynamics of the Activator pattern.

2.3 Structure of the Activator Pattern

A *client* is an application that uses services to perform portions of its computations. It accesses the services remotely using *service proxies*, which are obtained from an *activator*.

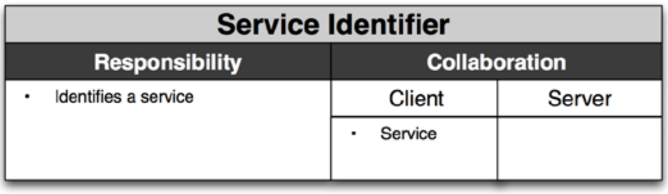


Client

Responsibility	Collaboration	
	Client	Server
• Uses services to perform portions of its computation • Accesses services via proxies • Obtains service proxies from the Activator	• Activator • Service Proxy • Service Identifier	• doWork()

For instance, in our example industrial automation system system, clients access services within the embedded controllers by connecting to these devices remotely. Example clients include material flow controllers that identify optimal paths for delivering goods to their destinations and administration consoles that monitor and control an automation system.

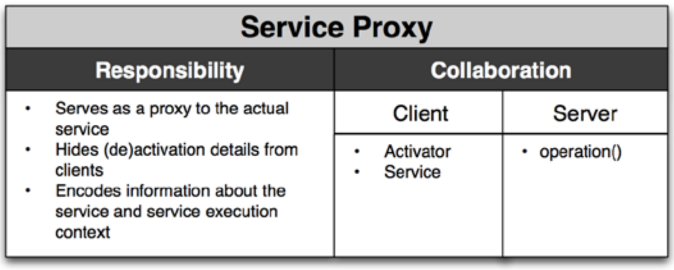
A *service identifier* is some type of entity, such as a web service universal resource locator (URL), CORBA interoperable object reference (IOR), or COM+ moniker, that clients use to identify a particular service. A service identifier can be created by either a server/service proxy or a client. A client passes a service identifier to an activator, which extracts the information required to locate and provide the requested service.



Service Identifier

Responsibility	Collaboration	
	Client	Server
• Identifies a service	• Service	

For instance, in our automation example, the service identifier is an IOR that opaquely encodes addressing information for a single service, including the host address of its embedded device, the port number on which an activator listens for incoming requests, and additional context information, such as the particular object that implements the service and its security credentials.



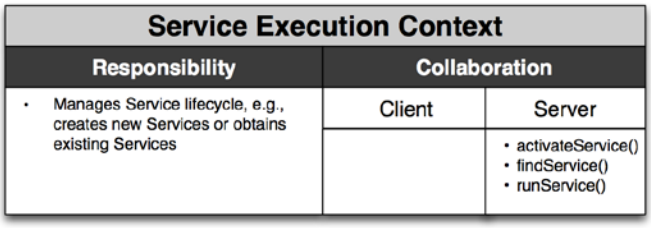
Service Proxy

Responsibility	Collaboration	
	Client	Server
• Serves as a proxy to the actual service • Hides (de)activation details from clients • Encodes information about the service and service execution context	• Activator • Service	• operation()

A *service proxy* is a proxy [POSA1, GoF] that resides with the client and facilitates its communication with the activator and service. It also shields clients from an activator's involvement in connecting clients and services. In addition, a service proxy can encode information about the service identifier, service, and the service execution context that can be used to optimize communication and enhance availability. The service proxy can either be an explicit proxy with concrete operations (as in the case of CORBA or EJB) or it can be more implicit (as is the case with web clients that activate HTTP servers by establishing TCP/IP connections).

For instance, in our automation example, the service proxy is an explicit proxy object that shields the client from system-level details of communication and activation. The service proxy uses the service identifier to extract the host, port, and other context information needed to direct client requests to their destinations.

A *service execution context* runs on a server, executes services, and controls their activation and deactivation lifecycles. Lower-level service execution contexts include operating system processes (which provide the unit of memory protection and resource allocation) or threads (which provide the unit of execution for instructions within a process). Higher-level service execution contexts include containers in component middleware that provide the context for processing operation invocations on components. Container-based service execution contexts often provide a factory (e.g., using the Factory Method or Abstract Factory patterns [GoF]) to create services and/or lookup functionality to obtain existing services.

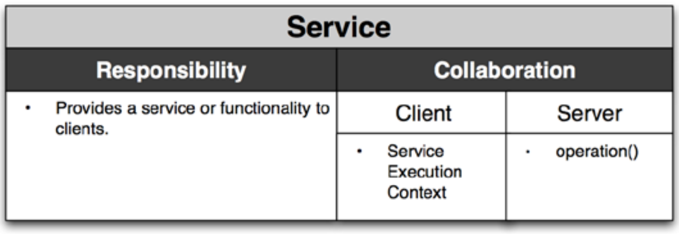


Service Execution Context

Responsibility	Collaboration	
	Client	Server
• Manages Service lifecycle, e.g., creates new Services or obtains existing Services		• activateService() • findService() • runService()

For instance, our automation example uses thread-based service execution contexts to run automation services implemented as C++ objects. After activating a service, the service execution context invokes a method on the service to initialize itself.

A *service* is an entity that runs on a server, is executed in a service execution context, and provides functionality and resources to clients. Services are named by their service identifiers and accessed by clients via their service proxies. A service must be registered with an activator manually by users or by some administrative entity.

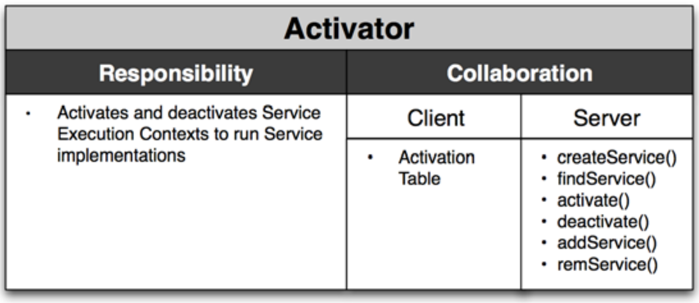


Service

Responsibility	Collaboration	
	Client	Server
• Provides a service or functionality to clients.	• Service Execution Context	• operation()

For instance, in our automation example, embedded system controllers provide remotely accessible services, such as command and control functionality that allows administrators to check and change the current system configuration. These service instances run in threads and consume various system resources, such as main memory, CPU time, sockets, or database connections. Multiple clients access these service components at various frequencies, i.e., not all services are accessed all the time.

An *activator* is a mediator [GoF] between services and their clients. It may run on each server or may be shared by a group of servers, but in either case it activates service execution contexts on demand. The activator uses an *activation table* to insert and remove registration information about services and their associated service execution contexts. When a client needs to access a currently inactive service, the activator activates a service execution context and arranges for the service to process the client's request(s) in this context.

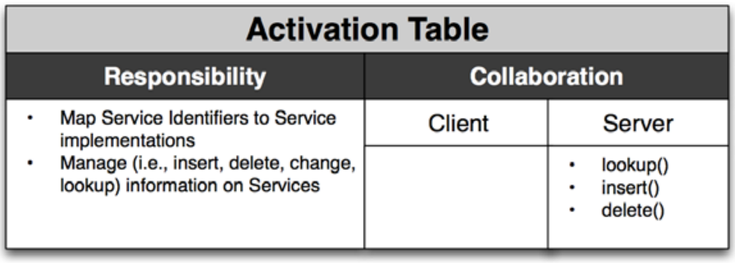


Activator

Responsibility	Collaboration	
	Client	Server
• Activates and deactivates Service Execution Contexts to run Service implementations	• Activation Table	• createService() • findService() • activate() • deactivate() • addService() • remService()

A client obtains a service proxy from the activator, which it then uses to invoke operations on the service. The activator uses information in its activation table to activate the appropriate service if it is currently inactive. Clients that query the activator for a service must indicate the desired service via a *service identifier*, which the activator uses to find the associated entry in its activation table.

For instance, activation in our automation example can involve different activities. An activator can be implemented as a remote gateway listening on a network port for incoming client requests. A client request is typically initiated via a service proxy. If the service's execution context has already been created, the activator simply forwards the client request to the service. If the service execution context has not been activated, however, the activator creates a thread to execute the service and initializes the service. After this initialization phase, the service proxy on the client is associated with the service execution context and the client request is forwarded to the service transparently.



Activation Table

Responsibility	Collaboration	
	Client	Server
• Map Service Identifiers to Service implementations • Manage (i.e., insert, delete, change, lookup) information on Services		• lookup() • insert() • delete()

An activator uses its *activation table* to map service identifiers to service implementations and service execution contexts. An activator uses this table to store associated registration and deregistration information when new services become available. These entries may include the execution path of the service executable or DLL, a reference to the service's interface, activation policies, and other configuration information.

For instance, the activation table in our automation example is implemented by a hash table that maps service identifiers to associated information, such as the port address of the service execution context, the address of the external service interface, a binary executable that implements the concrete service, a flag indicating whether the service execution context and the service are currently running, and other bookkeeping information.

The UML class diagram shown in Figure 2 depicts the relationships between the Activator pattern participants described above.

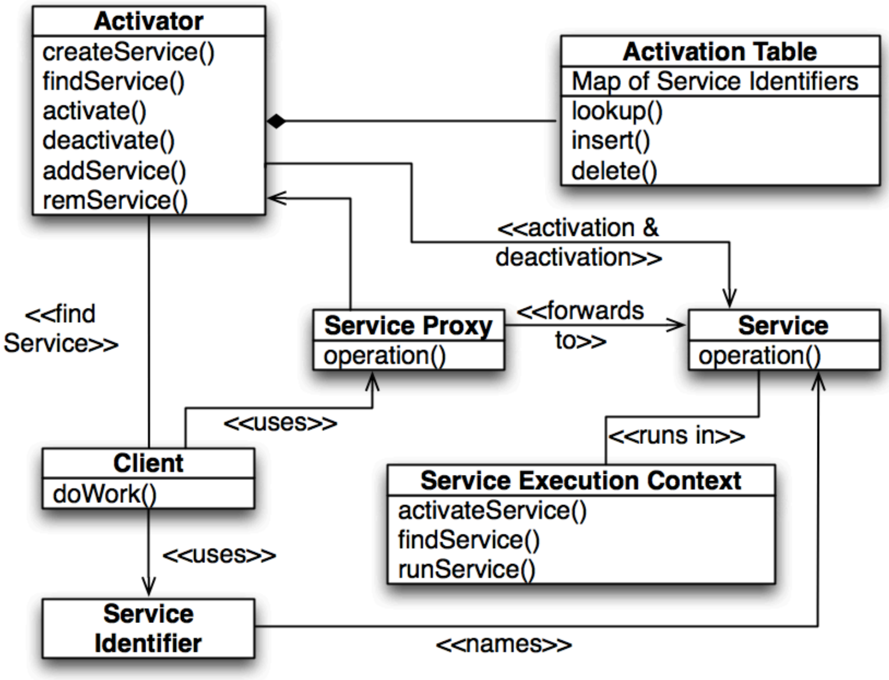


Figure 2: Roles and Responsibilities in the Activator Pattern

2.4 Dynamics of the Activator Pattern

There are three phases to the dynamics in this pattern: service registration, service activation and access, and service deactivation, as discussed below.

A. Service registration. This phase involves the following two steps that are shown in Figure 3:

1. A service developer implements a service using appropriate programming language and platform libraries or middleware.
2. The service is registered with the activator, which keeps track of where to locate the service implementation and under what conditions to activate it.

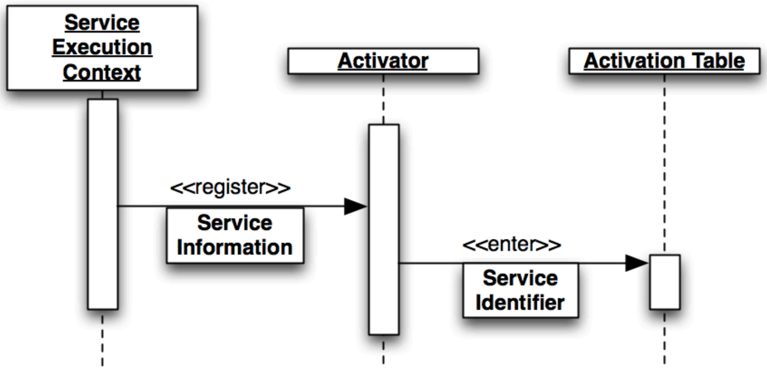


Figure 3: Service Registration Phase.

Service registration is discussed further in design and implementation activity 3.1 in Section 3.

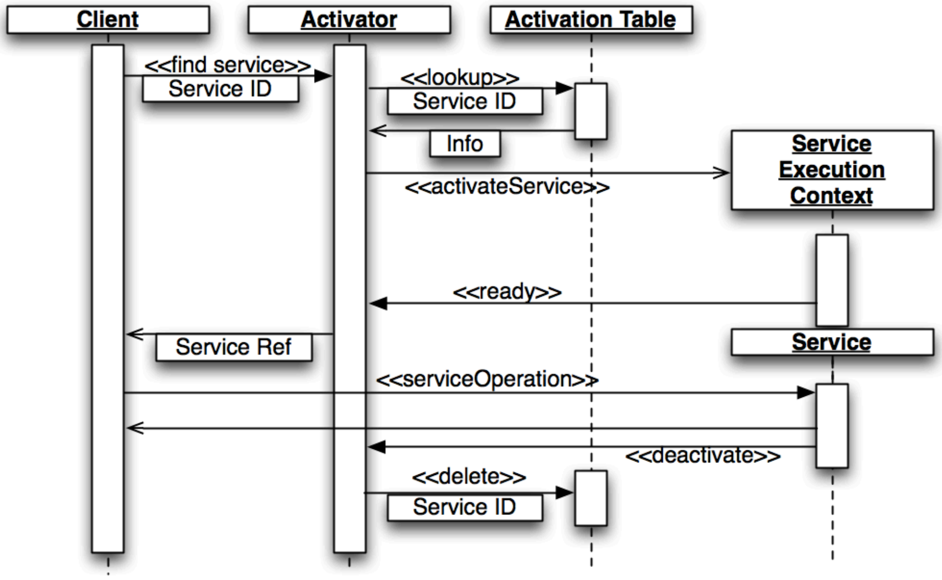


Figure 4: Service Activation, Access, and Deactivation Phases.

B. Service activation and access. This phase involves the following six steps shown in Figure 4:

1. A client uses the service's identifier to obtain a reference to a service, e.g., it can locate the reference in a naming service with a service identifier.
2. The client then invokes an operation on the service via its reference.
3. The client's request is first sent to the activator, which determines the service from the identifier in the request and finds the corresponding entry in the activation table.
4. The activator checks whether a service execution context running the service is currently active. If it is inactive, the activator uses activation-related information in its activation table to activate the service execution context that runs the service.

5. The activator waits for acknowledgement that the service execution context and the service it implements are activated and ready to receive requests.
6. The activator then transparently delegates the request to the service execution context, which performs the client's request and returns a reply if necessary.

Other aspects of service activation and access are discussed in implementation activity 3.2.

C. Service deactivation. The service deactivation process involves the following two steps also shown in Figure 4:

1. A service can be deactivated when no clients are accessing it.
2. Deactivation may cause the service to store any non-volatile state information in persistent storage and then terminate the service execution context it is running in.

Service deactivation strategies are discussed in design and implementation activity 3.3 in Section 3.

3 Activities Associated with Designing and Implementing the Activator Pattern

There are many ways to instantiate the Activator pattern. The following activities focus on key design and implementation issues associated with instantiating Activator in resource-constrained service-oriented systems.

1 *Define the services and service identifiers.* The services provided by a service-oriented system are usually specified in a requirements or system architecture/design document. If this information is not readily available, conduct domain analysis [Evans03, CLF93] to determine the types of services that applications will need. This domain analysis typically follows the process described in the Domain Object pattern [POSA4], which encapsulates each distinct unit of application functionality into a self-contained service. Likewise, representations of service identifiers are also often defined in various specifications or requirements documents. If not, consider using well-known service identifier representations, such as URLs, IORs, or TCP/IP port numbers and network addresses.

For instance, embedded system controllers typically require services for configuring, monitoring, and effecting system components in the automation domain.

2 *Identify services that should be activated and deactivated on demand.* For this activity, iterate through the following subactivities:

2.1 *For each service determine the costs of activating and deactivating services on demand versus keeping them alive for the duration of the system.* These costs are typically measured by instrumenting test programs that are representative of the services and running experiments to profile and quantify the resources used by the services on a particular operating system platform [DynamicProfiling08]. For this pattern to be effective, the time/space overhead used to activate services should be significantly lower than the time/space resource consumptions of the services that are activated.

For instance, although an embedded controller contains limited resources, such as CPU time or memory, monitoring services typically incur high usage of both resources. In contrast, activation time is relatively low (essentially the time needed to spawn a thread), so it makes sense to implement on-demand activation strategies for embedded controller services that do not have stringent real-time requirements.

2.2 *Determine client/service usage profiles and identify any stringent real-time requirements that may impede the use of the Activator pattern.* If instances of a particular service are used continuously throughout the whole lifecycle of their clients—and/or if it is critical that clients have low and predictable latency—they may not be good candidates for on-demand activation. For example, on-demand activation may be unsuitable for services with stringent real-time properties, such as a real-time controller for an anti-lock braking system with bounded latency and jitter requirements. Conversely, on-demand activation is well-suited for service accessed by clients sporadically, such as an FTP or HTTP service that lacks stringent real-time requirements. Another part of the service usage profile is how many instances of a given service must be active—and thus competing for the same resources—at the same time.

2.3 *Identify services for on-demand activation*. Using the results of previous subactivities, determine all services subject to on-demand activation, which typically have the following properties:

- They are used temporarily—not continuously—by clients, so it makes sense to activate/deactivate them on-demand to minimize resource consumption.

- The costs for activating and deactivating these services are negligible compared with the QoS requirements of clients, as well as with the time periods when these services must be available.

For instance, no services in our automation system example have stringent real-time requirements, so they are all candidates for on-demand activation via the Activator pattern.

3 *Develop a service activation and deactivation strategy*. For every service, determine the details of service activation and deactivation by performing the following subactivities:

3.1 *Define the service execution context representation and associated service registration strategy*. A service execution context can be implemented in various ways and at various levels of abstraction, including:

- *Lower-level service execution context*, such as an operating system process or thread.
- *Higher-level service execution context*, such as a container in component middleware, which provides the runtime context for a service implemented as a component.

The type of execution context representation selected typically dictates the service registration strategy. For example, the UNIX *Internet daemon* (Inetd) superserver [SteRago08] uses a text file called inetd.conf to define the Internet services that will be registered and activated by Inetd. Conversely, containers in component middleware typically have well-defined—often standard—APIs and protocols for registering services implemented as components.

For instance, our automation system implements service execution contexts using threads. All registration information, such as the factory for creating service implementations, is specified in a text file read by the Activator when it starts running.

3.2 *Define the service activation and access strategies*. There are several dimensions to this implementation activity, including:

- *Define the service initialization strategy*. If all services are stateless, little or no initialization may be required when activation occurs. If they are stateful, however, they must be initialized when they are activated. In some cases, the activator or the service execution context can handle initialization issues, e.g., an activator can invoke internal initialization methods of the service based on information stored in its activation table. In some cases, a service may perform its own initialization. In yet other cases, clients may be responsible for initializing their services.
- *Define the request delegation strategy*. After the activator has initialized the service, the client request must be delegated to it. There are two general delegation strategies:
  - *Server-mediated delegation*, where the activator simply forward the request to the service. The benefit of this approach is that there's no extra communication between the server and the client, i.e., the request is processed directly. The downside of this approach is that a client who converses with the same service for multiple requests will have to send each request through the activator .
  - *Client-mediated delegation*, where the activator sends back information to the client that updates the service proxy to point to the activated service. The benefit of this approach is that conversational clients can cache the updated service proxy and use it to optimize subsequent communication with the activated service. The downside is that the first request will incur extra communication back to the client before being forwarded to the service running on the server.

Broker pattern implementations [POSA1] often apply these delegation strategies with the broker playing the role of the activator.

For instance, in the automation example, all services are stateless so initialization is simplified and self-contained. Since clients often communicate with the same service for an extended period of time the client-mediated delegation strategy is used.

3.3 *Define the service deactivation strategy*. There are several strategies for deactivating services:

- *Service-triggered deactivation*. In this strategy, a service decides to deactivate itself, e.g., a service could deactivate itself if a designated period of time elapsed without any clients sending the service requests. This strategy is commonly known as the Evictor pattern [POSA3, HV99].
- *Client-triggered deactivation*. In this strategy, a client explicitly invokes an operation to trigger service deactivation. To implement client-triggered deactivation, the service must be notified

whenever a client obtains or releases its reference to this particular service. Internally, the service may keep a reference count that it increments/decrements on service access/release. When the count reaches zero, the service could deactivate itself to release its resources.

- *Activator-triggered deactivation*. In this strategy, the activator decides when to deactivate a service. For example, the activator might track resource usage on a particular computing node and deactivate services after a certain threshold is reached. Naturally, care must be taken to deactivate services gracefully to avoid disrupting vital processing and losing important state information.

In most cases, once the service is ready for deactivation it should inform its execution context so any resources allocated to the service can be released. The subsequent behavior of the execution context will depend on how it is represented. For example, if the service is implemented as a component and service execution context is implemented as a container, the container will delete the memory allocated to the component. Likewise, if the service is implemented within an OS process, the process may simply exit, thereby releasing the memory resource automatically.

For instance, our automation example uses service-triggered deactivation via the Evictor pattern, i.e., services deactivate themselves and terminate their service execution context if they do not receive any client requests after a certain period of time.

4 *Define the interoperation between services and the service execution context.* The service execution context may provide operations to (1) access information and resources managed by the execution context, (2) request service deactivation, and (3) modify the behavior of the service manager. Likewise, services might provide (1) global operations for service instantiation or (2) callback operations that the services execution context invokes automatically upon the occurrence of certain service lifecycle events, such as service creation/activation and deactivation/destruction.

For instance, services in the automation example implement a callback interface that is invoked automatically by the service execution context before a service is created and activated, as well as before it is deactivated and destroyed. The services use these callback methods to acquire or release resources.

5 *Implement the activator. This step involves the following subactivities:*

5.1 *Determine the association between activators and services*. There are a number of ways to associate activators and services, including:

- *Singleton activator*. Use the Singleton pattern [GoF] to make the activator a singleton and have all services share it within a particular environment, such as a process or a computing node. In this approach, an activation table keeps track of the services controlled by the activator.

- *Exclusive activator.* Provide each service or service execution context with its own activator. In this approach, an activation table can be used as a global repository accessible by all activator instances. The advantage of this approach is its higher scalability and reliability. Activator instances must coordinate access to the activation table, however, which can increase complexity.

- *Distributed Activator*. This approach generalizes the singleton activator. A local activator is placed on each computing node. When a client asks for a particular service, the local node's activator checks whether the corresponding service is available locally or remotely. In the former case, the workflow continues as in the singleton activator. In the latter case, however, the local activator determines where the appropriate service is available and then connects to the remote activator, on that computing node, which retrieves a reference to the service and returns it to the local activator. The local activator then returns the service proxy to the client.

For instance, in the automation example, the activator implementation uses the singleton activator approach. The singleton activator instantiates the appropriate service on demand when a new request arrives for any service provided by a computing node.

5.2 *Determine the degree of transparency*. There are various degrees of transparency from the client's perspective, including:

- *Explicit activator*. In some implementations of the Activator pattern, clients or their service proxies may be aware that they are retrieving services via an activator. In this case, an activator is a separate component that clients can contact explicitly to activate a service. The activator could also invoke the service and return the result to the client. Examples of explicit activation include network and system management systems, where administrators use management consoles to activate services on remote clients. In these systems, remote management agents provide management interfaces that contain operations for starting and stopping services explicitly to reduce resource contention on managed objects. In this context, management agents play the role of explicit activators.

- *Transparent activator*. It is often beneficial to shield clients from the activator, so they appear to access the service directly rather than indirectly via the activator. To implement a transparent activator, therefore, the Interceptor pattern [POSA2] can be used to contact the activator before the service is created. For example, an EJB or CCM container uses an interceptor to activate components on demand. Likewise, CORBA's General Inter-ORB Protocol (GIOP) provides a special message (LocateRequest) that an Implementation Repository activator uses to intercept client requests, create service execution contexts on demand, and redirect clients to the newly activated service.

As explained in implementation activity 3.3, an activator implementation should work together with services and/or service execution contexts to cleanup resources when services are deactivated.

For instance, in the automation example, the activator implementation uses the Interceptor pattern. Whenever a new request arrives, the communication framework notifies the activator, which then instantiates the appropriate service on demand and deactivates it later using the Evictor pattern.

6 *Define the contracts between interoperating participants*. A contract specifies the set of interfaces implemented by each pair of parties that communicate and protocols they must obey. The protocol can be modeled with Activity or interaction diagrams and interfaces can be modeled with class diagrams.

First, determine the internal contracts that are not visible to clients, such as:

- *The contract between the activator and the service execution context*, which specifies how an activator locates, registers/unregisters, and (re)activates a service, as well as (re)activates and registers/unregisters services managed by the service execution context. This contract can also limit the number of copies of a service execution context that an activator should activate, which can be used to prevent intentional or accidental denial of service attacks.
- *The contract between the service execution context and its services*, which introduces interfaces for creating, initializing, and releasing services. It also specifies how a service can notify its service execution context about its deactivation.

Second, define the external contracts that are visible to clients, such as:

- *The contract between the client and the activator*, which defines how a client obtains a service proxy from the activator. This contract defines a service identifier that encapsulates addressing information for the service and service execution context where the service implementation runs. An activator knows how to extract this information from a service identifier.
- *The contract between the client and the service*, which defines (1) the set of operations that a client can use to access the functionality of the service via its service proxy and (2) the means of disconnecting from and/or deactivating the service after its processing is complete. The service proxy is often implemented as a proxy that exposes this contract via explicit operations, as is the case with CORBA or EJB. It is possible, however, to implement this contract implicitly via lower-level means, such as TCP/IP connections or protocol messages, as is the case with Internet services implemented via HTTP, FTP, and SSH daemons.

For instance, the stateless instances of services in our automation example system are created by the service execution context on demand and deactivated using the Evictor pattern [POSA3]. The service execution context is implemented as a remote object that the activator contacts to forward client requests. Since services are stateless, there is a 1:1 mapping between service execution requests and services, which simplifies the interface between the activator and the service execution context. The eviction strategy is configured statically into the system. All service instances are preinstantiated and organized in a pool that can shrink or increase as required.

The interface between clients and activators is straightforward. Clients obtain service identifiers from a database. Service proxies are instantiated from a client-side library, passing the service identifier as an argument. The service proxy implements the service interface and shields clients from lower-level network programming details. The service proxy sends requests to the activator and passes results back to clients, thereby shielding clients from changes to the activator implementation. For instance, while subsequent versions of the automation system might configure each service execution context to use thread pools to pre-instantiate groups of service instances, clients will not be affected by these changes.

In addition to the design and implementation steps covered above, the following are also several common software design variants of the Activator pattern:

- *One service per service execution context.* Instead of allowing a service execution context to provide multiple service types, this variant enforces a 1:1 relationship between service execution contents and services. Each service execution context implements exactly one service. The advantage of this approach is the reduced complexity of the activator implementation. Resource contention increases, however, when more service execution contexts are available. This approach is therefore most useful when services have a long execution time or when the number of services is relatively small.
- *Combined Component Configurator and Activator*. This pattern compound [POSA5] combines the Component Configurator pattern [POSA2] with the Activator pattern to enhance both on-demand resource allocation *and* flexibility. In this variant, an activator is responsible for activating/deactivating service execution contexts in which services run, whereas a component configurator is responsible for determining which service implementations are actually linked into a server from a dynamic link library (DLL). This pattern compound yields a flexible design with well-defined separation of concerns. For example, the activator in this approach could spawn a process to serve as the service execution context and then use a component configurator within the process to link service implementations from DLLs into the process when the process is launched by the activator.

4 Case Study Results and Consequences

This section presents a case study that applies the Activator pattern to the industrial automation system summarized in Section 1, quantifies the resource utilization impact of the Activator pattern, and generalizes the benefits and liabilities of applying this pattern to the case study.

4.1 Applying the Activator Pattern to an Industrial Automation System

Applying the Activator pattern as described in Section 3 improved the scalability of the industrial automation system [POSA4] by ensuring that memory resources are consumed only by services being accessed by clients. Figure 5 shows how activating services on demand improves system scalability.

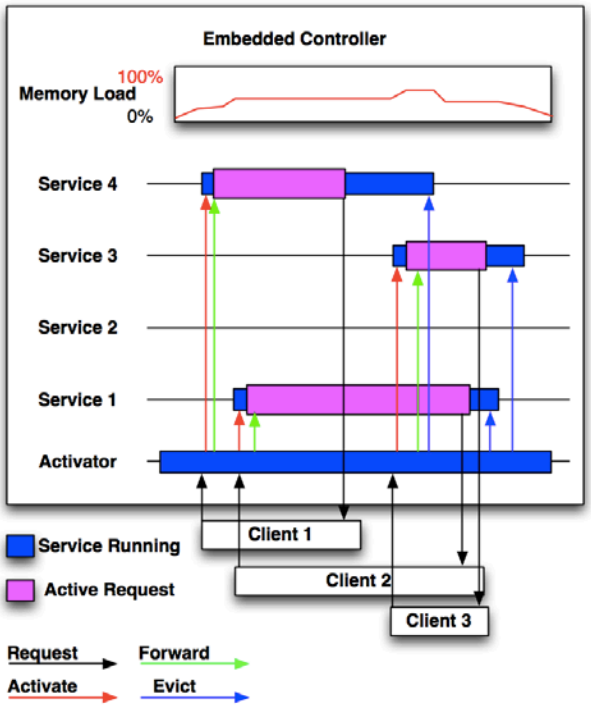


Figure 5: Improving System Scalability via the Activator Pattern.

In the initial implementation of the industrial automation shown in Figure 1 in Section 1, only a small number of clients could access the system simultaneously since scarce system resources were devoted to running unused services that consumed memory. In the refactored implementation shown in Figure 5, however, a larger number of clients can access the same (or different) services simultaneously without incurring overload. Even three clients concurrently accessing the embedded controller do not incur more than 50% memory load, so the overload threshold will not be reached.

After refactoring of the initial eager resource allocation strategy, the revised system uses a singleton activator to create service execution contexts and activate services on-demand. It uses the Evictor pattern to deactivate services when clients do not access them after a designated period of time. Some addition runtime overhead is caused by the activator spawning threads to run newly activated services, but this overhead is negligible since each client exchanges a number of requests with the service before focusing its attention elsewhere.

To quantify the resource utilization impact of the Activator pattern, we conducted an experiment that simulated a logging system [BeautifulCode07] portion of an industrial automation system that consisted of hundreds or thousands of sensors and actuators. In this context, a process-per-connection concurrency strategy may be desirable to meet reliability and fault tolerance requirements, i.e., messages indicating faults in the system must be delivered reliably so portions of the embedded controllers can be modified to account for the failure and accompanying loss of functionality. The logging server for this experiment used the Activator pattern described in Section 3 to spawn new logging processes on demand, rather than the eager allocation strategy described in Section 1 that maintained a persistent process for each sensor/actuator.

The logging Activator was installed on a virtual machine running the Fedora 15 Linux distribution. Other virtual machines on the same host were configured to establish connections to the logging server at a rate of ~100 hertz and deliver logging messages of variable size from 50 bytes to 500 bytes. The Activator was instrumented via high-resolution timers to determine (1) the amount of time necessary to launch a logging service process so it is ready to accept an incoming log message and (2) the amount of time necessary for the Activator itself to accept a new incoming connection attempt. The average amount of time required for the Activator to spawn a new process and be ready to accept a new connection was 748 microseconds with an individual footprint of 32 kbytes per instance. While 32 kbytes is a relatively small footprint per connection, this would require several megabytes of resident memory footprint for deployments consisting of hundreds or thousands of concurrent connections, which would increase the resources required for the industrial automation system.

4.2 Generalized Consequences of Applying the Activator Pattern

The Activator pattern offers the following benefits when applied to resource-constrained service-oriented systems, such as the industrial automation case study presented in Section 4.1:

- *Scalable resource usage.* Service execution contexts only run when services are being accessed by clients. They are deactivated and reactivated on demand, which helps improve the scalability of the overall system by allocating resources more parsimoniously.
- *Implicit initialization.* All details of service and service execution context activation and deactivation are encapsulated by the activator interface, which enables service developers to initialize services when they are activated. For example, service state can be stored in a database and loaded whenever the service execution context is activated, so clients may not need to initialize services explicitly themselves.
- *Exchangeable strategies due to transparent service creation.* As a consequence of using activators as intermediaries, the service creation strategy can be exchanged without impacting clients. For example, an activator can choose between different services supporting the same service type via load balancing or fault tolerance replication mechanisms.
- *Location transparency with respect to services.* If the service proxies returned by the activator point to proxies, the location of the service can be made invisible to clients. Clients can thus access services residing on remote machines transparently.
- *Efficient and fast service access.* After clients have obtained updated service proxies from an activator, they can access the services directly, bypassing further indirection and delegation.

The Activator pattern, however, also has following liabilities:

- *QoS penalties due to activation overhead.* When a client first accesses an inactive service, the activator must activate a server execution context to run the service, which increases the latency and jitter of the initial access. It is also possible for clients to trigger intentional or accidental denial-of-service attacks by activating many services unnecessarily.

- *Complex state management.* If service execution contexts running services are deactivated and activated on demand, any non-volatile state must be persisted across succeeding passivation and activation events, which can complicate service development.
- *Debugging and testing can be hard*. Decoupling clients from the activation of services can make it harder to determine why failures occur. For example, if there is not enough memory to activate a service in a service execution context, the client may not be able to ascertain what caused the problem since service activation is supposed to be transparent.
- *Limited domain of applicability*. The Activator pattern focuses on the efficient management of resources such as memory, CPU, and batter power. It does not, however, address efficient management of network resources, bandwidth, scaling services across network/network topology considerations, or high availability requirements.

4 Related Work

The Activator pattern has been applied to many service-oriented systems, some of which are described in this section.

Object Request Broker (ORB) and Component Middleware frameworks, such as CORBA, CORBA Component Model (CCM), Microsoft COM+, and Java RMI use the Activator pattern in several ways. For example, they use the pattern to transparently spawn server processes when clients invoke operations on remote objects, as follows:

- In COM+ the Service Control Manager (SCM) can spawn server processes on demand. It then connects to the appropriate class factory and creates a new instance of a COM object. The activation table is implemented by a combination of the Windows registry and internal tables. A global DLL, called OLE32.DLL encapsulates access to the activator implementation transparently for clients.
- CORBA ORBs use *transparent activators* to activate servers on demand. When a client invokes an operation on an object reference, the call initially goes to an Implementation Repository [VH99], which plays the role of the activator in this pattern. The Implementation Repository checks to see if a server process containing the object being accessed by the client is running. If it is not running, the server process is spawned. After the Implementation Repository verifies the process is running, it returns a LOCATION_FORWARD exception to the client ORB, which updates the object reference to note the new location and reissues the call to the server transparently to the client application.

Component middleware uses the Activator pattern to activate components transparently via a hierarchy of activators. For example, in the CORBA Component Model (CCM) the Implementation Repository is used to spawn server processes. Servant activators can then be used to create containers that provide the runtime environment for managing the lifecycle of component implementations. Similar mechanisms are available in Enterprise JavaBeans.

OS superservers. The Activator pattern has been used in OS ‘superservers’ that manage network servers. Two common OS superservers are Inetd [SteRago08] and Listen [Rago93], which consult configuration scripts that specify (1) *service names*, such as the standard Web and Internet services HTTP, TELNET, FTP, DAYTIME, and ECHO, (2) *port numbers* to listen on for clients to connect with these services, and (3) *an executable file* to invoke and perform the service when a client connects.

Both Inetd and Listen contain a master acceptor process that monitors a set of port numbers associated with the services. When a client connection occurs on a monitored port, the acceptor process accepts the connection and demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service, either reactively, proactively, or as an active object [POSA2], and returning results to the client as needed.

Smartphone service activators. Since memory, power, and CPU processing are relatively constrained on smartphones (such as Google Android, Apple iOS, and Nokia Symbian), it is common for these smartphones to automatically activate and deactivate applications to ensure there are sufficient resources available to meet user needs. As the user launches applications, they are loaded into main memory from disk. To ensure maximum responsiveness as users switch between applications, they remain loaded in memory after the user exits the application. Since these platforms do not swap memory pages to disk, the physical memory space eventually becomes exhausted. When this happens, the operating system will first request that running applications release memory, then begin to forcibly evict applications from memory [AppleDev].

Web servers use the Activator pattern to start services on demand when HTTP requests arrive. Plug-ins may be registered with the Web server (e.g., using configuration files or Component Configurators [POSA2]), which represent service execution contexts. These plug-ins handle HTTP requests for specific URL addresses. For example, when a URL specifies a file with a PHP file-extension, a PHP-plug-in is accessed by the web server to handle this kind of request. Handling the request in this context means to load the PHP interpreter, execute the PHP-script specified, and return an HTML page to the originator of the request. To optimize performance, the server only activates plug-ins on demand when an appropriate request arrives.

Human usage. A human known use of the Activator pattern is a call center used to provide technical help desk services, credit card fraud reporting, or airline reservations. Here the resources to be optimized are telephone lines, computer and database connections, and call center operators. The activator is the central system that is called by customers. After a customer has specified their service identifiers via voice or touchtone input, the call center activator connects the customer to the appropriate operator, after first activating the resources needed by the operator to handle the call, which can involve establishing network and database connections, preparing information on the user interface display, etc. The customer is then connected directly to the operator. Hanging up the telephone triggers service deactivation and releases the allocated resources for use in servicing other customer calls.

The Activator pattern is also related to other patterns that have been documented in the pattern literature, as described below.

The *Component Configurator* pattern [POSA2] allows applications to dynamically link and unlink their component implementations at run-time without having to modify, recompile, or statically relink application code. The primary difference between Component Configurator and Activator is that Activator focuses on activating/deactivating a service execution context on-demand, whereas Component Configurator focuses on dynamic linking/unlinking the code that runs within an execution context. The Component Configurator and Activator patterns can be combined to form a compound pattern, as described at the end of Section 3.

The *Virtual Component* [PLoP9] and *Virtual Proxy* patterns [POSA1] can also be used in conjunction with the Component Configurator pattern to provide an transparent way of loading and unloading components that implement middleware and/or application software functionality. These patterns ensure that the software provides a rich and configurable set of functionality, yet occupies main memory only for components that are actually being used. Whereas the Virtual Component and Virtual Proxy patterns focus largely on creating component memory on demand, the Activator pattern focuses on a broader set of issues, such as locating services and activating/deactivating service execution contexts on demand.

The *Service Activator* pattern [J2EEPatterns03, EIP03] can be used to receive asynchronous client requests and messages. When a message is received, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously. Whereas the Service Activator focuses on invoking business methods on business service components, the Activator pattern's focus is broader and also includes launching processes and/or threads to perform arbitrary services.

The *Broker* pattern [POSA1] structures distributed software systems with decoupled components that interact via local and/or remote invocations. A broker component is responsible for coordinating communication, such as establishing connections and forwarding requests, as well as for handling results and exceptions. Remote objects represent services that reside in servers. For performance and scalability reasons, these Broker systems often instantiate the Activator pattern to spawn server processes on demand. A common example is the Implementation Repository in CORBA-based ORBs [VH99].

The *Lazy Acquisition* pattern [POSA3] defers the acquisition of resources late in the system lifecycle, e.g., at installation- or run-time. Although this pattern is similar to the Activator pattern, these patterns address different problem contexts at different levels of abstraction. The Lazy Acquisition pattern defines a broad strategy for allocating resources, such as shared, passive entities like memory or connections, to active entities, such as services. Activator, in contrast, is a more focused pattern that addresses the activation and deactivation of service execution contexts and services in resource-constrained service-oriented computing environments.

The *Eager Acquisition* pattern [POSA3] describes how run-time acquisition of resources can be made predictable and fast by eagerly acquiring and initializing resources before their actual usage. This pattern is the opposite (aka, "pattern complement" [POSA5]) of the Activator pattern (and Lazy Acquisition pattern). In addition to focusing on the context of satisfy high predictability and performance in resource acquisition time, the Eager Acquisition pattern (like the Lazy Acquisition pattern) defines a broad strategy for allocating resources, such as shared, passive entities like memory or connections, to active entities, such as services. Activator, in contrast, is a more focused pattern that addresses the activation and deactivation of service execution contexts and services in resource-constrained service-oriented computing environments, where it is more important to conserve used (constrained) resources than to optimize runtime performance and minimize latency.

The small memory patterns in [SmallMemory] describe a range of other techniques that can be applied to reduce the consumption of memory in embedded systems and handheld devices with their limited computing horsepower.

5 Concluding Remarks

Unnecessary utilization of limited resources, such as memory, network connections, threads, and power, can degrade the overall quality of service of applications resource-constrained domains, such as embedded automation systems and mobile applications. In these domains, eager allocation strategies that allocate services and leave them running both before and after they are needed are often inappropriate since they can lead to resource exhaustion. A preferred solution is for the underlying operating system or middleware platform to evict services that are not in use to free up resources for active application services.

If an evicted service is subsequently accessed by its clients, however, it must be re-activated, which can involve recreating the resource, reloading its state, restarting it in its server, and reacquiring any resources it in turn uses. Such service-oriented resource reactivation should ideally be transparent to client and server applications so it appears as if the accessed resources were always available. Likewise, applications services should ideally be shielded from managing such activation and reactivation explicitly.

The Activator pattern described in this paper provides an efficient and transparent means to automatically minimize resource consumption by activating services on demand and deactivating services when they are no longer being accessed by clients. Our empirical tests compare the time/space tradeoffs between the Activator pattern's "lazy allocation" model vs. alternative "eager allocation" models. These results are consistent with other studies [PerfPatterns98, PerfPatterns99, PerfPatterns11] that measure the performance implications of alternative patterns.

References

[AppleDev] Apple iOS Developer Library, *The Application Runtime Environment*, Apple Inc., 2011.

[BeautifulCode] W. Otte and D. Schmidt, "Labor-Saving Architecture: an Object-Oriented Framework for Networked Software," *Beautiful Code*, edited by Greg Wilson, O'Reilly, 2007.

[CLF03] D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[EmSOA] D. Barisic, M. Krogmann, G. Stromberg, and P. Schramm, "Making Embedded Software Development More Efficient with SOA," In International Conference on Advanced Information Networking and Applications Workshops (AINAW), pages 941–946. IEEE Computer Society, 2007.

[DynamicProfiling08] D. Waddington, N. Roy, and D. Schmidt, ``Dynamic Analysis and Profiling of Multi-threaded Systems," *Designing Software-Intensive Systems: Methods and Principles*, Edited by Dr. Pierre F. Tiako, Langston University, OK, April, 2008.

[EIP03] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003.

[Evans03] E. Evans, *Domain-Driven Design*, Addison-Wesley, 2003.

[Fayad03] M. Fayad, H. Hamza, and H. Sanchez, "A Pattern for an Effective Class Responsibility Collaborator (CRC) Cards, The 2003 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, October 2003.

[Fayad04] M. Fayad, H. Sanchez, and H. Hamza, A Pattern Language for CRC Cards, the 11th Pattern Language of Programs (PLOP) September 8-12, 2004, Allerton Park, Monticello, Illinois, USA.

[Fowler03] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, (3rd ed.). Addison-Wesley, 2003.

[GoF] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[HV99] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

[JavaRMI] W. Grosso, *Java RMI,* O'Reilly, 2001.

[J2EEPatterns03] D. Alur, J. Crupi and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd Edition, Prentice Hall June, 2003.

[PerfPatterns98] D. Petriu and X. Wang, "Deriving Software Performance Models from Architectural Patterns by Graph Transformations," In Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg (Eds.). Springer-Verlag, London, UK, 1998.

[PerfPatterns99] G. Somadder, D. Petriu, "Pattern-guided Performance Measurements of OO Multi-threaded Servers," Proceedings of the ISCA 13th International Conference on Computers and Their Applications (CATA-98), Honolulu, Hawaii, USA, March 25-27, 1998.

[PerfPatterns11] N. Mani, D. Petriu, and M. Woodside, "Towards Studying the Performance Effects of Design Patterns for Service Oriented Architecture," Proceeding of the second joint WOSP/SIPEW International Conference on Performance Engineering (ICPE '11). ACM, New York, NY, USA, 2011.

[PLoP9] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual Component: a Design Pattern for Memory-Constrained Embedded Applications," Proceedings of the 9th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, September, 2002.

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1 – A System of Patterns,* John Wiley & Sons, 1996.

[POSA2] D.C. Schmidt, M. Stal, H. Rohner, F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2 – Pattern for Concurrent and Networked Objects*, John Wiley & Sons, 2000.

[POSA3] M. Kircher and P. Jain: *Pattern-Oriented Software Architecture, Volume 3 - Patterns for Resouce Management,* John Wiley & Sons, 2004.

[POSA4] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture, Volume 4 – A Pattern Language for Distributed Computing*, John Wiley & Sons, 2007.

[POSA5] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture, Volume 5 - On Patterns and Pattern Languages*, John Wiley & Sons, 2007.

[Power07] G. Creus and M. Kuulusa,M, "Optimizing Mobile Software with Built-in Power Profiling," in *Mobile Phone Programming and its Application to Wireless Networking*, F. Fitzek and F. Reichert, Eds. Springer, 2007.

[Rago93] S. Rago: UNIX System V Network Programming, Addison-Wesley, 1993.

[SmallMemory] C. Weir and J. Noble, *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, 2000.

[SOAP] E. Newcomer, *Understanding Web Services*, XML, WSDL, SOAP, and UDDI, Addison-Wesley, 2002.

[SteRago08] R. Stevens and S. Rago, *Advanced Programming in the UNIX Environment*, Addison Wesley, 2008.