On Mechanisms for Deadlock Avoidance in SIP Servlet Containers

Y. Huang, L. K. Dillon, and R. E. K. Stirewalt

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
{huangyi7,ldillon,stire}@cse.msu.edu

Abstract. Increasingly, VoIP applications are being deployed to multi-threaded SIP servlet containers. Unfortunately, the standard specification for these containers is silent with regard to synchronization issues, and the lack of direction has led different vendors to adopt a variety of different and incompatible policies for preventing data races among concurrent threads. In addition to the obvious portability problems, some policies make servlet code prone to deadlock under common scenarios of use. This paper documents this problem with concrete examples and proposes modifications to the standard to allow programmers to implement the protocols needed to avoid these common deadlocks. These extensions will open the way for automatic generation of the synchronization logic needed to implement these avoidance protocols, thereby increasing the safety and reliability of applications deployed in this environment.

Key words: Concurrency, converged container, deadlock prevention, negotiation, servlet, SIP, synchronization contract

1 Introduction

A typical Voice-over-IP (VoIP) application comprises multiple, dynamically configurable services (i.e., features), such as *call waiting*, *call forwarding*, and *call merge* [8]. Such services are usually implemented atop a signaling protocol, called the session initiation protocol (SIP) [20], for setting up and tearing down media sessions. To simplify their implementation and composition, these services may be implemented as *SIP servlets* and deployed to *SIP application servers* that reside in nodes across the Internet. The JSR 116 standard defines the API for programming SIP servlets [15]. A key concept in this standard[1] is a middleware abstraction called a *container*, which automates the handling of various concerns, thereby separating them from the "business logic" of a service. This paper explores one of these concerns—thread synchronization—which, to date, has proved difficult to relegate to the container.

[1] and also in application-server architectures, e.g., Apache Tomcat [26], WLSS [1], and Sailfin [23].

To demonstrate the difficulties, we show how the policies adopted by one popular container are prone to deadlock, and discuss problems of using vendor-specific APIs to implement common deadlock avoidance techniques. This demonstration provides motivation for prescribing extensions to the SIP Servlet API that enable separation of concurrency concerns and automated deadlock prevention. We then outline a method by which such extensions can be achieved. For brevity, this paper discusses only deadlock avoidance, although the general approach also provides for avoidance of critical races. In a nutshell, the idea is to encapsulate a deadlock avoidance protocol within *negotiators*, which are specialized *concurrency controllers* [6] for SIP threads. We further advocate that negotiators should be generated automatically from high-level *synchronization contracts* [4, 25].

The need to synchronize threads executing within a container arises because, upon receiving a message, the container dispatches a dedicated thread to process the message. As a result, multiple threads processing different messages may attempt to concurrently access the same session data. Additionally, a thread may create and use data that persists in the server beyond the thread's lifetime and, consequently, that may be accessed by multiple threads. For instance, various kinds of *session* objects are used to maintain the evolving state of a call [2]. To prevent data corruption, containers employ some policies for concurrency control, typically locking these shared objects prior to processing the message.

Unfortunately, these *container-level synchronization policies* are rarely documented, and current standards documents (e.g., [15] and [27]) are silent on the issue. Container operations implicitly lock some resources but leave the application to explicitly lock others. Without knowing precisely what locks the container acquires and when it acquires them, application developers are ill equipped to judge whether the container will guarantee a given set of synchronization requirements or whether and how to develop custom synchronization logic to guarantee these requirements. In fact, results of experiments suggest that different vendors have adopted different container-level synchronization policies. This situation leads to synchronization-related failures, typically corruption of session data or deadlocks. Moreover, the synchronization policies adopted by some vendors make it impossible to avoid deadlocks in some situations for which a servlet could be designed to avoid deadlocks if the container did not implicitly acquire any locks on session data.

The remainder of the paper explores these issues in the context of a specific container—the container in a BEA WebLogic SIP Server (WLSS) [2]. We first present background on the context in which the problems arise and background relating to our proposed approach for addressing these problems (Section 2). We then provide concrete examples to illustrate how the container's synchronization policy leads to deadlock under two scenarios of use that occur in practice (Section 3). The key contributor to these deadlocks is that some resources are implicitly acquired and held whereas other resources must be explicitly acquired during the processing of a message. In one of these scenarios (Section 4.1), deadlock can be avoided by using non-standard operations in the

WLSS API. But this deadlock avoidance method does not scale, limits reuse, and is prone to error. In the other scenario (Section 4.2), deadlock cannot be prevented in a WLSS container. However, a mechanism proposed for the new standard, JSR 289 [27], which is currently under public review, can be used to implement a common, albeit somewhat crude, deadlock avoidance heuristic. We discuss problems and limitations of this mechanism for avoiding deadlock. The issues raised by these "work-arounds" motivate extending the SIP Servlet[2] API with facilities for avoiding deadlocks by a technique that separates concerns involving deadlock avoidance from the servlet's "business logic" (Section 5). Finally, we summarize and identify directions for future research (Section 6).

2 Background

For concreteness of exposition, we consider the synchronization policy and implementation adopted by one vendor—BEA WebLogic SIP Server (WLSS) [1]. Thus, by way of background, we briefly describe the synchronization policy implemented in a WLSS (Section 2.1). The motivation for a container-supplied synchronization policy is to simplify the programming of applications and make them less prone to synchronization-related errors. This goal is achievable only to the extent that (1) the synchronization logic can be cleanly separated from the "business logic" of an application and (2) the synchronization code can be automatically generated. We thus briefly overview the related work on separation and automatic programming of synchronization concerns (Section 2.2). Finally, we provide some background on synchronization contracts, which the approach suggested in Section 5 builds on (Section 2.3). Synchronization contracts are designed for separation and automated enforcement of synchronization concerns in a special class of systems, which includes telecommunications systems.

2.1 The BEA WLSS

The WLSS policy mandates that the thread handling a message must have exclusive access to the *call state* associated with the message. Intuitively, call state comprises the persistent data pertaining to a call, which might be consulted or modified by the thread processing the message. Unfortunately, we could not find a precise definition of call state in the WLSS documentation. This situation is problematic, as application developers need to know precisely what constitutes a call state in order to create applications that do not exhibit data races, deadlock, or other synchronization errors. In fact, the WLSS policy leads to deadlock in some contexts, as we demonstrate in subsequent examples.

Lacking a clear definition of call state, we take it to mean at least the *SIP application session* associated with the message. A SIP application session (hereafter *SAS* for brevity) is an instance of the SIP Servlet API class `SipApplicationSession` and is used to encapsulate and provide uniform access to the persistent

[2] Follow the convention in [2], "SIP Servlet" (uppercase "S") refers to the standard API, while "SIP servlet" refers to a program that uses the API.

data for an instance of an application. We know that the WLSS policy guarantees the thread handling a message will have exclusive access to the SAS associated with that message. Thus, when illustrating problems that derive from this policy, we use examples whose synchronization requirements are limited to the sharing of SASs rather than the data that these SASs encapsulate.

WLSS is a *SIP and HTTP converged container*, which means that it provides for processing of both SIP messages and HTTP messages. It contains a single *message handler*, which hosts a dedicated thread, and zero or more SIP and/or HTTP threads. The message handler listens to the network for incoming messages. When a message arrives, it dispatches the message to a thread and then returns to listening. It dispatches SIP messages to SIP threads and HTTP messages to HTTP threads. Based on information in the dispatched message, a SIP thread selects an appropriate *SIP servlet* and invokes the `service` operation on this servlet, passing the message as an argument. When the invocation of `service` returns, the thread terminates. For HTTP messages, WLSS performs the same steps, except that it selects an HTTP servlet instead of a SIP servlet. A SIP application comprises at least one SIP servlet, zero or more HTTP servlets, and supporting resources (e.g., JSP pages and images).

A key difference between how a SIP thread processes a message and how an HTTP thread processes a message is that a SIP thread (implicitly) performs synchronization operations, whereas an HTTP thread does not. A SIP thread acquires a lock on the SAS associated with the message prior to invoking the `service` method and releases the lock once the invocation returns. Consequently, any operation invoked during an activation of a SIP servlet's `service` operation may freely access the SAS associated with the message without concern for data races or consistency of transactions. For brevity in the sequel, we refer to the SAS associated with the message a thread is processing as the *thread's SAS*.

The SIP Servlet API provides operations using which a SIP thread may access and modify sessions, including SASs associated with messages that might be being processed by other SIP threads. In WLSS, the implementations of some of these API operations contain synchronization logic over and above the implicit SAS locking functionality provided by the container when servicing an incoming message. More precisely, an API operation that accesses a SAS embeds code to lock the SAS at the start and to release the lock on the SAS just before returning, unless the SAS is already locked by the thread that invokes the API operation (as is the case for the thread's SAS).

2.2 Approaches to separation and automatic programming of synchronization concerns

The separation and automatic programming of synchronization concerns serves as the motivation for many concurrent programming frameworks, including D [16], concurrency controllers [6], SyncGen [10], Szumo [3, 25], and Java transactions [13, 14]. Separating the synchronization logic from the "business logic" is intended to simplify programming both concerns by virtue of not needing to "mix" the two. However, separated or not, synchronization protocols can be difficult to implement correctly. Thus, automatic programming of the synchronization logic is preferred to leaving this responsibility to the application programmer. Of particular concern in this paper are synchronization protocols that provide mutually exclusive access to sets of shared resources and the deadlock problems that easily arise when these protocols are not correctly implemented.

Concurrency controller is a design pattern that generally supports the separation and automatic generation of complex synchronization protocols. The name of the pattern was first suggested by [6], whose system of the same name uses the pattern. In the sequel, we use the term to refer to the general style of concern separation embodied in the pattern as opposed to the particular framework in [6]. A concurrency controller can be thought of as a generalization of a *monitor* that can be made to apply to multiple resources. Application code invokes an operation to lock a set of resources, understanding that the call will block if the controller cannot successfully acquire the whole set. When the call returns, the application assumes exclusive access to these resources. The API to a controller is very simple, with operations designed to resemble event notifications or actions. However, the operations might implement complex collaborations with other controllers to acquire a set of resources while avoiding deadlock. Approaches that employ this pattern include SyncGen, which generates them from declarative *region invariants* [10], the system of Betin-Can and Bultan, which generates them from action-language specifications [6], and our own Szumo system, which generates them from declarative *synchronization contracts* [25, 5].

Unlike WLSS, where the container serves as a gatekeeper to the SASs, a concurrency controller does not monitor when each thread attempts to access which resources. Instead, it provides a public controller interface, which publishes available actions that a thread could perform on the shared resources under its control as well as acceptable patterns of use—i.e., usage obligations and exclusion guarantees. In multi-threaded systems that are designed according to the pattern, a thread consults the controller before it performs actions on any of the shared resources under the controller's control. A key feature of this approach is that the application programmer is responsible for invoking operations on the controller to obtain permission to access shared resources.

Finally, we should mention that concurrency frameworks that support *transactional memory* sidestep the deadlock problem by avoiding the use of locks altogether [13]. Rather than locking shared resources, a thread performs operations on copies of the shared resources and then attempts to reconcile conflicting updates when the transaction commits. While an elegant abstraction, especially in the context of data-oriented enterprise applications, transactions must be able to be aborted (and the changes rolled back or discarded). This assumption is not reasonable in the communications domain because operations cannot in general be rolled back (if, for instance, an operation involved the issuing of a message over the network) and because application sessions might be much too large to clone. To some degree, the need to clone large shared objects can be ameliorated using *transaction synchronizers*, which permit multiple transactions to operate on the same objects [17]. Still, the resulting transactions must be abortable, which again, is not generally feasible in this domain.

2.3 Synchronization contracts

Our ideas for deadlock-avoiding SIP servlets build on a model of synchronization for object-oriented (OO) programs, called Szumo [3], which leverages synchronization contracts to support component-based development of a limited class of multi-threaded OO applications. Termed *strictly exclusive systems*, this class comprises applications in which threads compete for exclusive access to dynamically changing sets of shared resources. In addition to telecommunication applications, examples include extensible web servers and interactive applications with graphical user interfaces. This narrowing of focus was motivated by the observation that many applications fit well in this category and that, in such cases, we can exploit a clean, compositional model of synchronization.

Szumo adopts the principles of Meyer's design-by-contract approach to reliable software [18]. It associates each thread with a synchronization contract, which describes client and supplier rights and responsibilities when performing operations that use shared resources. The contracts themselves are formed by instantiating and conjoining module-level *synchronization constraints*, which a programmer declares in the modules' interfaces. For example, a synchronization constraint for a servlet that implements a call-merge service might declare that, when processing a request to merge two calls, a SIP thread needs exclusive access to the SASs associated with the calls. The application developer is responsible for verifying that a SIP thread accesses these SASs only under conditions covered by a synchronization constraint. The developer then writes the servlet code assuming exclusive access to the SASs under these conditions; i.e., she does not write explicit synchronization logic to acquire and release the SASs. In this way, synchronization contracts provide a useful separation of concerns.

In lieu of synchronization logic, synchronization contracts are automatically enforced and negotiated at run-time. Contract models, such as SCOOP [18] and Szumo [4, 3, 25], assign much of the responsibility for contract negotiation to a run-time system or a middleware, which schedules processes based on deadlock- and starvation-avoidance heuristics. We previously integrated Szumo into an extension of the Eiffel language [3, 25] and, more recently, implemented it in C++ as a framework [11]. A case study in evolving and maintaining a multi-threaded web server demonstrates how the use of synchronization contracts supports maintenance activities [5]. The work described in the current paper shows how synchronization contracts can be used to make container-level synchronization policies more flexible with regard to how, when, and how many objects are locked during the servicing of a message. Containers that employ these more flexible policies are able to avoid a large class of deadlocks, some of which cannot currently be avoidable in containers that employ more rigid policies.

[3] for the SynchroniZation Units Model; an early version of this model was called the "universe model" [4]

3 Example deadlocks

To illustrate how deadlock can arise in a SIP container, we present two scenarios that arise in practice. The first involves a Call-Merge service, which is a SIP service used in conferencing applications (Section 3.1). The second involves a Click-To-Dial service, which is a SIP and HTTP converged service (Section 3.2). We then briefly reflect on the fundamental problem illustrated by these examples (Section 3.3).

3.1 Deadlock in a Call-Merge service

A Call-Merge (CMG) service merges two calls for use in implementing a conference call. A *CMG request message* designates two calls: a *source call*, involving the user agent requesting the merge, and a *target call* that is to be merged. If the source and target calls belong to different application sessions, the thread that processes the request message must lock multiple SASs, and deadlock could arise if two or more concurrent threads each attempts to lock these SASs in a different order. Such a situation arises in practice, e.g., when two user agents, each involved in a call, both try to merge the call involving the other agent at nearly the same time.

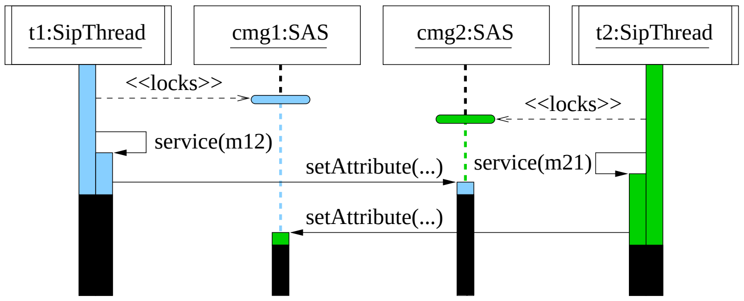

Fig. 1. Sequence diagram illustrating a deadlock in the CMG scenario

Figure 1 depicts a deadlocking call-merge scenario using extensions to UML 2.0 sequence diagrams to visualize key aspects of thread and synchronization state. A filled activation bar[4] on the lifeline of a thread represents an activation of that thread's run method. Shading distinguishes activations that are executed by different threads (different shades) and activations that are blocked (filled black). To indicate the acquisition of a lock on a shared object, we use a dashed arrow, labeled by the stereotype `<<locks>>`, that emanates from an activation and ends in a shaded bubble that is centered on the shared object's lifeline. Moreover,

[4] called an *execution specification* in UML 2.0 [22].

we shade the lifeline of a shared object that has been locked according to the shading of the activation that locked it. Finally, we depict operation invocation and return in the usual manner with one exception: When a thread invokes an operation on a servlet, we depict that activation as a nested activation of the thread object. Thus, in Figure 1, activations of the servlet's `service` operation abut the activations of the run operation of the two threads, $t1$ and $t2$, that invoke `service`. Without this convention, our diagrams would need to explicitly depict the servlet object, which would then need to host multiple, concurrent, activations of `service` operations by different threads.

Prior to this scenario (and thus not depicted in the figure) the container received two CMG request messages—$m12$ requesting to merge source call $c1$ with target call $c2$, and $m21$ requesting to merge source call $c2$ with target call $c1$. Suppose call $c1$ belongs to SAS $cmg1$ and $c2$ belongs to SAS $cmg2$. Upon receiving $m12$ and $m21$, the container dispatched two threads, $t1$ and $t2$, to handle them. In the scenario, each thread first locks its own SAS, as required by the WLSS synchronization policy, and then invokes the `service` operation on the CMG servlet. Each thread then proceeds to process its message, during which it must access the SAS for its target call. In this scenario, $t1$ invokes the API operation `setAttribute` on $cmg2$. Because $t1$ does not hold the lock on $cmg2$, the `setAttribute` operation under WLSS tries to acquire this lock. This attempt causes $t1$ to block because the lock is already held by $t2$. Likewise, $t2$ blocks when it invokes `setAttribute` on $cmg1$. At this point, deadlock ensues.

3.2 Deadlock in a Click-To-Dial service

A Click-To-Dial (CTD) service allows a subscriber to initiate a call from a web browser by selecting or entering a party to call, hereafter the *callee*, and then clicking on a "dial" button. Clicking this button triggers the browser to send a *CTD request message*, an HTTP message designating the subscriber and the callee, to a CTD service. When the container receives this message, it dispatches the request to an HTTP thread, which then attempts to establish two SIP calls—one between the CTD service and the subscriber and the other between the CTD service and the callee. The HTTP thread attempts to establish these calls by creating and sending out two SIP invitation messages and by creating a new SAS that will encapsulate any relevant call state associated with these messages and with the SIP calls that are ultimately established.

The HTTP thread also creates an instance of a class called `callManager`, which is designed to allow the subscriber to monitor and affect the state of a call from her web browser. The call manager is used, for instance, to perform management tasks such as pausing the SIP calls if the subscriber's balance drops to zero and then prompting her to deposit new funds via a web interface. To properly reflect the state of a call, the call manager must be notified once the SIP calls are established. To enable this notification, the HTTP thread registers the call manager as a *listener* of the SAS associated with the call. A SIP thread processing a message associated with this SAS may then notify the manager to signal changes in the state of the call. Because the call manager may need to be accessed by multiple threads, including both HTTP and SIP threads, class `callManager` is implemented as a monitor.[5]

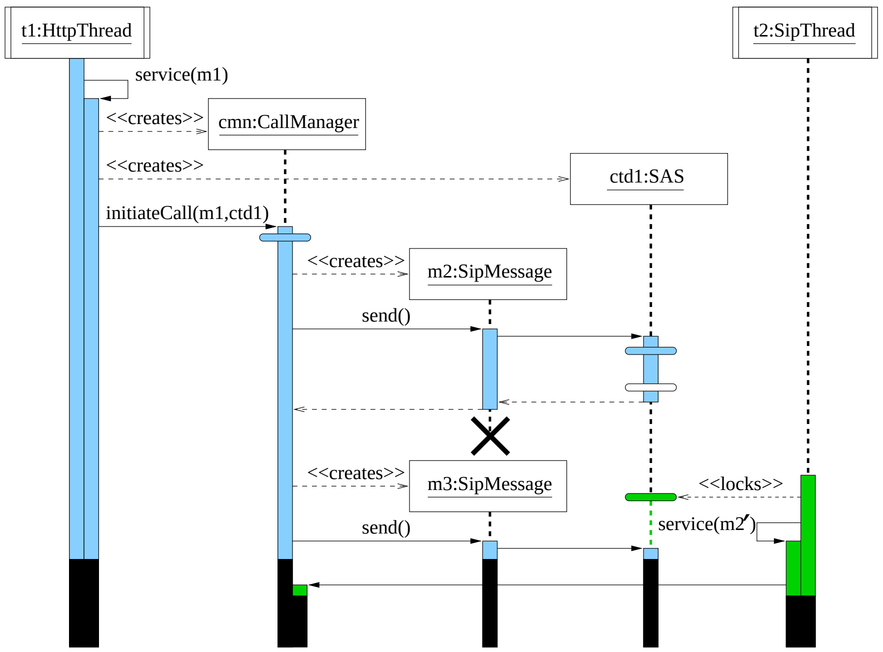

Fig. 2. Sequence diagram illustrating a deadlock in the CTD scenario

Figure 2 illustrates a deadlock that may arise when using the CTD service. Prior to the start of this scenario, the container dispatches the HTTP thread $t1$ to process the CTD request message, $m1$. In processing $m1$, the HTTP thread first creates a call manager object, *cmn*, and a new SAS, *ctd*1. It then invokes an operation on *cmn* to initiate the call requested by $m1$. As part of initiating the call, the call manager registers itself with *ctd*1 (not shown) and then creates and sends the two SIP invitation messages $m2$ and $m3$. When sending a message, the container locks the SAS associated with the message. Thus, calls to `send` will proceed only if the lock on the SAS can be acquired. To reduce clutter, when an operation on an object locks or unlocks the object itself, we omit showing dashed arrows and stereotypes. A shaded bubble on the activation signifies that the lock is acquired and a clear bubble signifies that the lock is released. Thus, in this scenario, the activation of `send` on $m2$ succeeds in acquiring the lock, and so $m2$ is sent. However, the activation of `send` on $m3$ blocks because the concurrent thread $t2$, which is dispatched with the response $m2'$ to the invitation message

[5] by means of the idioms in Java, its methods are declared to be `synchronized`.

$m2$, acquires the lock on $ctd1$ before cmn invokes the `send` operation on $m3$. Subsequently, when $t2$ attempts to notify cmn (which is registered with $ctd1$ to be notified of responses), it also blocks because $t1$ is still executing within cmn (a monitor).

3.3 Deadlock prevention

Because JSR 116 is silent on issues of synchronization, container vendors are free to provide their own policies and primitives. Unfortunately, the WLSS policy of determining and then implicitly locking an application session based on the message to be serviced is prone to deadlock in common use cases, as illustrated by the CMG and CTD deadlock scenarios. Clearly, some means for deadlock prevention are needed. Deadlock-prevention strategies are generally well known and fall into one of two categories—*avoidance* and *recovery* [19]. In Szumo, we used a combination of these strategies to prevent a large class of deadlocks. Such strategies are key to the approach we describe in Section 5 and suffice to avoid the deadlocks illustrated by the CTD and CMG scenarios. However, before describing our approach, we now briefly describe how programmers can use existing (vendor-specific) APIs to implement common deadlock avoidance heuristics and discuss limitations of doing so. Recovery heuristics are not really possible under the WLSS policy, as there are no means for one SIP thread to force another to release its lock on a SAS.

4 Known work-arounds

Deadlock avoidance strategies use information about the resources that might be requested to develop a protocol for making requests. If every client requests its resources in the order prescribed by this protocol, then deadlock can be avoided. The CTD deadlock can be avoided in this manner by designing the HTTP thread to access the $ctd1$ application session indirectly through a *transaction proxy*, which allows a programmer some control over when an application session is locked and for how long. Currently, transaction proxies must be implemented using proprietary extensions to JSR 116 (Section 4.1). Not all deadlocks are avoidable using transaction proxies. In fact, the CMG deadlock scenario cannot be avoided by such means. However, the JSR 289 draft specification provides a feature called *session key based targeting* [27, § 15.11.2], which allows the programmer more control in selecting the application session to implicitly lock when a message is received (Section 4.2). Servlet programmers could use this feature to avoid the CMG deadlock; however, the feature was not designed for this purpose, and its use in this context is brittle, inflexible, and ultimately error prone. We now briefly delve into these workarounds to better illustrate the approaches and to motivate why we believe deadlock prevention should be automated whenever possible.

4.1 Extensions to support transactions

To implement a deadlock-avoiding request protocol, programmers must be able to control when a lock on an application session should be acquired. Unfortunately, such control is quite limited in the current WLSS implementation of JSR 116. When a SIP message arrives, the container determines the application session associated with this message and locks this session before ceding control to the servlet. If a servlet needs to access some other application session, the lock on this other session is acquired and released on demand, as the servlet invokes methods on that session. Those locks are necessarily acquired after the servlet has been made to hold the lock on the SAS associated with the message it is serving. Consequently, if a servlet requires access to multiple application sessions, these are necessarily locked and accessed one after another; they cannot be locked atomically[6] before being accessed.

To cope with this problem, WLSS provides proprietary extensions to program what we call transaction proxies, which are implemented using a combination of two design patterns—*proxy* and *command* [12]. A transaction proxy is a proxy that acts as a surrogate application session. It provides all of the operations of an application session but implements them by delegating to some other instance of class `SipApplicationSession`, i.e., the "real" application session. In addition, a transaction proxy provides a method called `doAction`, which is parameterized by an *activity*, i.e., a sequence of operations encapsulated into an object via the command pattern. When `doAction` is invoked with an activity, the transaction proxy locks the real SAS and then executes the `run` method of the activity. Because the proxy holds the lock on the real SAS while executing the activity, the sequence of operations within the `run` method execute as a single transaction on this SAS.

Transaction proxies allow the servlet programmer to affect the order in which some of the resources needed to execute a transaction are acquired. Figure 3 depicts how the CTD deadlock can be avoided by encapsulating the `initiateCall` operation into an activity (instance of interface `WlssAction`) which is supplied to the `doAction` method of a transaction proxy (instance of class `WlssSipApplicationSession`) to $ctd1$. Notice that both $t1$ and $t2$ now request cmn and $ctd1$ in exactly the same order, thereby avoiding deadlock.

Transaction proxies provide one way for programmers to implement deadlock-avoidance protocols in their servlet code. The approach works for the CTD scenario because of an inherent asymmetry in the problem: Because the container does not implicitly lock resources on behalf of the HTTP thread, that thread can use a transaction proxy to mimic the acquisition protocol of any SIP threads that might respond to the generated messages. Without this asymmetry, deadlock cannot be avoided using transaction proxies. In addition, because a transaction proxy locks only a single application session, if a servlet requires access to more than two application sessions, it will need to use nested transaction proxies. As nesting grows deeper, designs become more brittle and more difficult to extend.

[6] in the sense that either the servlet locks all of them without blocking or it blocks without holding any of them until such time as it is able to lock them all.

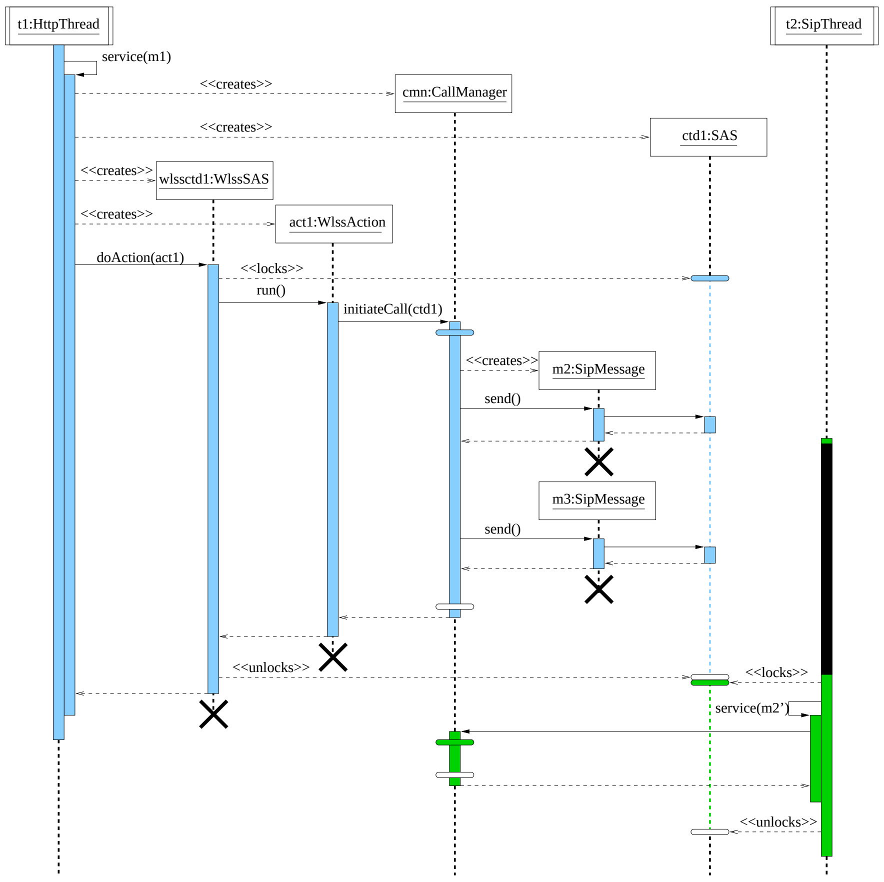

Fig. 3. Using WLSS proprietary API's to avoid the CTD deadlock scenario

Thus, when viewed as a mechanism for implementing deadlock-avoidance protocols, transaction proxies are limited in power and suffer problems of scalability.

Moreover, regardless of implementation mechanism, we contend that it would be poor policy to make servlet programmers responsible for developing and adhering to deadlock-avoidance protocols. Deadlocks are avoided only if every servlet deployed to a container respects the protocols that apply to the resources it needs. If a service that requested resources in violation of some protocol were ever deployed, it could cause deadlock in services that were designed correctly. Given the explosive growth and dynamic nature of VoIP services, developing, documenting, maintaining, and enforcing correct usage of these protocols would be a daunting task. For all of these reasons, we believe deadlock avoidance should be automated—either completely relegated to the container or automatically generated from higher-level specifications—but not programmed explicitly by application programmers.

4.2 Controlling target of implicit locking

While neither JSR 116 nor the WLSS extensions can be used to avoid the CMG deadlock, JSR 289 introduces a mechanism that could be used to avoid it. Recall that the WLSS container implicitly locks the application session associated with a message before any servlet code is executed. This implicit locking policy makes it impossible to implement deadlock-avoidance protocols, save for protocols in which the application session associated with the message is the first to be requested. In sharp contrast to the conventions of JSR 116, which specifies that an initial request message always results in creation of a new application session, the session key based targeting mechanism allows the programmer to select which application session should be associated with an initial request message. Using this mechanism, servlet programmers can implement deadlock-avoidance protocols involving arbitrary sequences of resource requests.

A programmer uses the targeting mechanism to associate a request message with a specific SAS using a static servlet-class method. This method is specially annotated in order that the container can differentiate it from other methods. The method so annotated takes a request message (i.e., a `SipServletRequest` object) as its argument and returns a *key* (string). Before dispatching an initial request message to a thread, the container checks whether the servlet it is routed to has such a static method. If so, it passes the message to this method and associates the message with the SAS whose identifier matches the returned key. If this method does not exist or if the returned key does not match that of any existing SAS the container creates one and associates it with the message.

To avoid the CMG deadlock, a programmer could use the targeting mechanism to associate both $m1$ and $m2$ with the same application session. Figure 4 illustrates how this mechanism works [7]. Assuming without loss of generality that the targeting mechanism associates both messages with $cmg1$, both $t1$ and $t2$ start by requesting the same application session. One of them (here, $t2$) blocks, allowing the other ($t1$) to acquire the other application session and proceed.

While the targeting mechanism supports the implementation of protocols that avoid CMG-like deadlocks, the solution is inelegant and inflexible. To compute which application-session identifier to return, the targeting method may need to query one or more application sessions. These queries would need to be threadsafe, which means one or more application sessions may need to be locked in order to compute which other application session(s) should be locked. In addition, the use of the targeting mechanism to synchronize request processing may impose unnecessary synchronization requirements on response processing. This occurs because, according to the SIP protocol, the application session associated with a request must also be associated with all responses to that request. Consider two request messages, $m1$ and $m2$, which under normal circumstances

[7] Here, we assume the static method is named `SipApplicationKey`.

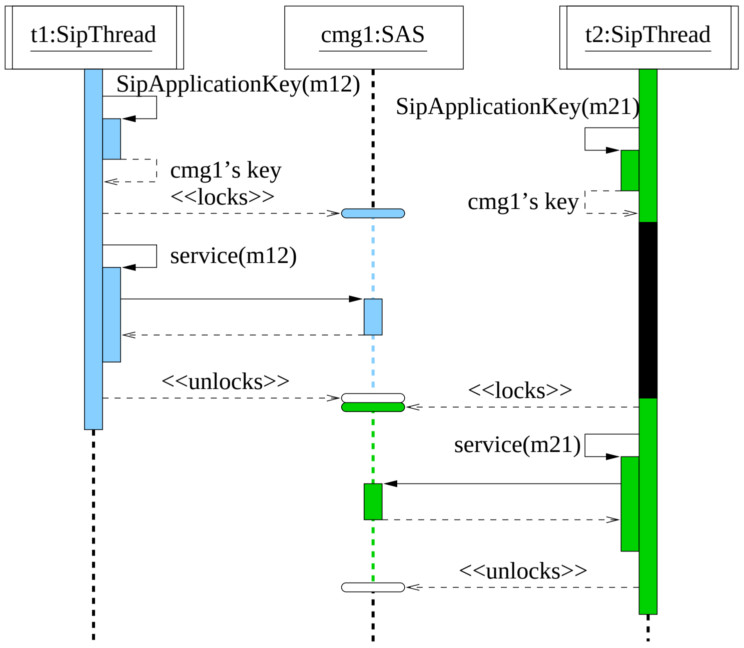

Fig. 4. Using JSR 289's session key based targeting mechanism to avoid the CMG deadlock scenario

would be associated with application sessions $a1$ and $a2$. Suppose further that, to avoid deadlock, the targeting mechanism is used to associate both messages with $a1$. The corresponding response messages, $m1'$ and $m2'$, will then necessarily be associated with $a1$, even though $m2'$ may only need to modify application session $a2$. That is, the container will artificially synchronize execution of threads that process response messages even if the resource needs of these responses do not overlap. Moreover, if processing response messages requires additional synchronization over and above that needed to process the initial request, the solution cannot be safely used.

5 Proposed extension

Clearly, the SIP Servlet API should be extended with facilities for avoiding the kinds of deadlocks mentioned previously. At a minimum, it might require a mode that performs no implicit locking so that servlet programmers can choose to use an approach such as we describe here. However, to foster the development of sound and maintainable servlets, facilities that separate concerns involving deadlock avoidance from the servlet's "functional logic" would be preferable. As mentioned previously (Section 2.2) the concurrency controller design pattern affords such separation by encapsulating complex synchronization protocols behind a simple API, which client code can invoke prior to entering a critical region. We now propose a modest extension to the SIP Servlet API to support the development of servlets that are designed according to this pattern (Section 5.1), and we illustrate how servlets so designed could avoid the aforementioned deadlocks (Sections 5.2-5.3). The complex synchronization logic used in this approach could be generated from abstract models of a servlet's synchronization states and transitions and a collection of declarative synchronization contracts (Section 5.4).

5.1 Proposed extensions

In our proposed extension, when the container allocates a thread to service a message, the container would create a *negotiator* object and pass that object along with the message to the `service` method. The negotiator object plays the role of a concurrency controller, where the thread executing the `service` method is the client program that uses the negotiator to guard entry into critical regions. We use the term *synchronization state* to refer to an abstract state of execution within which a servlet needs exclusive access to some set of resources. While servicing a message, a thread might cycle through several distinct synchronization states. Thus, before entering a new synchronization state, a thread would need to notify its negotiator of this intention. The negotiator, being a concurrency controller, would then negotiate for exclusive access to the resources denoted by the target synchronization state, blocking until such time as these resources can be acquired.

For this approach to work, a thread's negotiator must have been designed with knowledge of its owner's synchronization states. Moreover, because each synchronization state might denote multiple resources, each transition is a potential source of deadlock, which means the negotiator must implement an appropriate deadlock-avoidance protocol. To this end, the negotiator could potentially implement any of a number of different strategies; however, the negotiators for different servlets would need to agree on the strategy. For sake of generality, negotiators will likely need to implement sophisticated avoidance protocols, possibly involving some combination of simple resource numbering with age-based heuristics, e.g., wound–wait [21]. Fortunately, the details of these sophisticated protocols will be encapsulated within the negotiator and should not impact the design of the servlet code.

To accommodate this proposal, the SIP Servlet API would need to be extended to define a new abstract base class `Negotiator`, which a servlet programmer would extend to implement a negotiation protocol according to the synchronization states of the servlet being developed. Moreover, when the container creates a thread to service an incoming SIP message, it would need to analyze the message to choose an appropriate subclass of `Negotiator`.[8] The container

[8] Here, we assume the mapping of message type to Negotiator subclass could be specified in a deployment descriptor.

would then instantiate this class and pass the object along with the incoming SIP message to the servlet's `service` method. The servlet is then responsible for initializing this negotiator object and notifying it of synchronization state transitions when appropriate. In sharp contrast to the WLSS policy, the container does not acquire any locks implicitly before invoking `service`.

In order to separate out the details of synchronization from the servlet, the negotiator must know the synchronization states that are meaningful to the servlet, and it must know the resources needed in each state. Knowledge about the states and the names of the resources denoted by each state must be programmed into the negotiator. However, dynamic information, such as the actual resources that should be bound to these names, must be supplied at run time. For instance, suppose as in the CMG example, the CMG message encodes information about the source and destination calls. To access this information, the servlet will need to access the SIP application sessions (SASs) associated with each call. The negotiator will have been programmed using names that must be bound to these resources. The message should, therefore, be passed as a parameter to the constructor of the particular negotiator so that these SASs can be extracted from the message and bound to the names. Moreover, if the running servlet redirects a name to a new resource, it will need to communicate this change to the negotiator.

This proposal has the advantage of separating the logic of resource negotiation from the functional logic of a servlet; however, it requires that a servlet invokes a negotiator prior to entering a new synchronization state. This need to keep the servlet and its negotiator in sync suggests that negotiators (and the servlet-based notification code) should be generated automatically. Fortunately, negotiators can be automatically generated from high-level specifications, such as action languages [6], region invariants [10], and synchronization contracts [4, 25]. Synchronization contracts have the added advantage of being able to ensure freedom from certain kinds of data races [25]. We will therefore say more about generation from synchronization contracts in Section 5.4.

5.2 CMG scenario using negotiators

Figure 5 illustrates how a servlet designed using our proposed approach can avoid deadlock in the CMG example. As in the deadlocking scenario, the container dispatches two CMG request messages to two SIP threads. However, rather than locking application sessions $cmg1$ and $cmg2$ prior to invoking `service`, each thread creates a negotiator object and passes this object as a parameter to the `service` operation. Here, the code for `service` notifies its negotiator of its intention to transition to a new synchronization state. In this example, both threads attempt to enter a synchronization state called *merging*, which (for both threads) denotes the resource set $\{cmg1, cmg2\}$. In this case, the negotiator for $t1$ acquires both of these resources and returns, thereby allowing $t1$ to continue; whereas the negotiator for $t2$ cannot immediately acquire either resource. The latter thread therefore blocks until such time as it can acquire both resources.

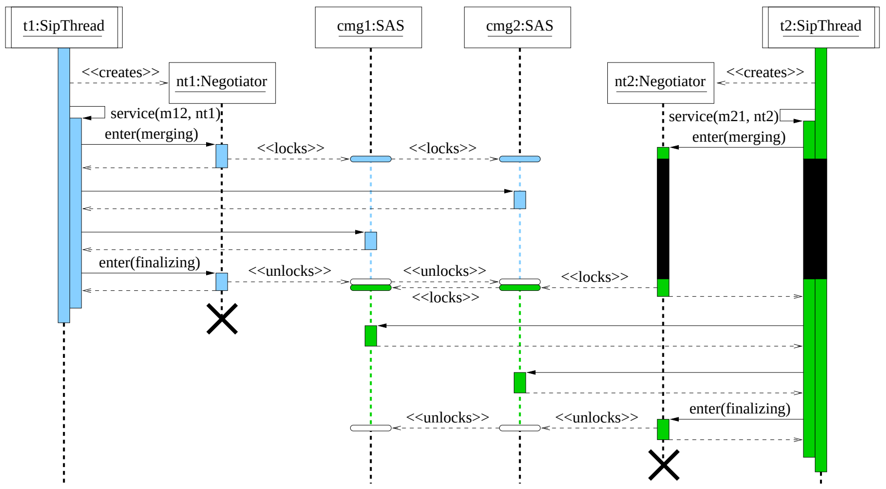

Fig. 5. Sequence diagram illustrating how negotiator use avoids the deadlock in the CMG scenario

During negotiation, the negotiators use a deadlock-avoidance strategy, the details of which are elided here for brevity.

Deadlock is avoided using this solution because the container made no implicit locking decisions prior to calling `service` and because the designer(s) of these negotiator objects were able to formulate a request protocol that avoids deadlock. Moreover, unlike the solutions presented in Section 4, all of the details of deadlock avoidance are relegated to the negotiators. A servlet merely needs to notify its negotiator when it transitions to a new synchronization state.

5.3 CTD scenario using negotiators

Figure 6 illustrates how our proposed approach avoids deadlock in the CTD example. As in the deadlocking scenario, the container dispatches an HTTP thread to service the incoming message. As before, the first actions of the `service` method are to create *cmn* and *ctd*1. However, before invoking `initiateCall` on *cmn*, the servlet notifies the negotiator of its intention to enter a synchronization state called *initiating*, which denotes the resource set $\{cmn, ctd1\}$. Having been notified, the negotiator proceeds to acquire these resources while avoiding deadlock.

In this scenario, $t1$ continues to hold the lock on *ctd*1 for the duration of its transaction. Thus, when $t2$ is dispatched to service $m2'$ and attempts to enter its *accepting* state, which denotes $\{ctd1, cmn\}$, $t2$'s negotiator cannot acquire the lock on either resource. At this point, $t2$ blocks. Eventually $t1$ transitions to a synchronization state called *finalizing*, which denotes the empty set of resources.

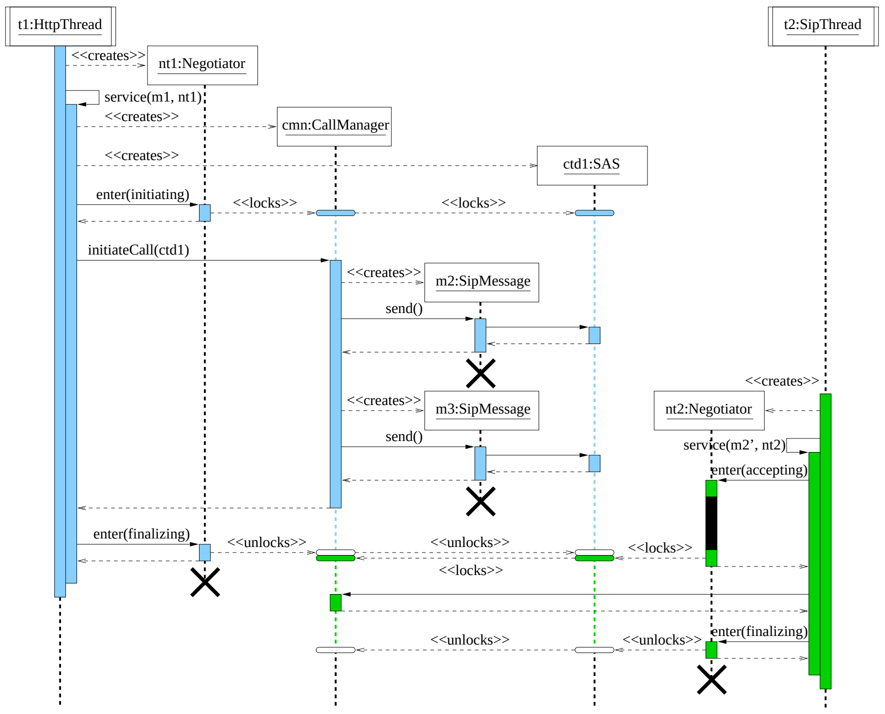

Fig. 6. Sequence diagram illustrating how negotiator use avoids the deadlock in the CTD scenario

At this point, $nt1$ releases its locks on cmn and $ctd1$, after which $t2$ is awakened, acquires these resources, and continues. Deadlock is therefore avoided.

5.4 Automated generation of negotiators

The effectiveness of our approach in practice relies on the following:

– the negotiator must be designed with knowledge about the synchronization states of the corresponding servlet,
– the servlet must notify the negotiator of state changes and communicate other information, such as the run-time identity of resources to bind to names, and
– negotiators must agree on the protocol used to acquire resources.

If these tasks were left to the programmer, negotiator development would be highly error prone.

Fortunately, our prior experience integrating synchronization contracts into object-oriented languages leads us to believe that both negotiators and also the logic needed to instrument a servlet to notify/update a negotiator can be generated automatically [25, 5, 11]. Two inputs would be required:

– a suitably precise model of a thread's synchronization states and transitions, and
– a collection of *synchronization contracts*, each of which maps a synchronization state to a collection of *resource names.*

A resource name is an identifier or a navigation expression, such as might be expressed using Blaha's object-navigation notation [7], which refers to a resource. Resource names can be viewed as program variables or expressions that show how to traverse a sequence of links starting from some named program variable. In the CMG example, the resource names `source` and `target` refer to SIP sessions, which maintain call state needed for the servlet to communicate with the endpoints of the call. By contrast, the resource names `source.SAS`, and `target.SAS` are navigation expressions that specify how to retrieve the application sessions associated with these SIP sessions. Resource names must be *bound* to resources at run time. By virtue of name binding, each contract can be interpreted as mapping a synchronization state to its denotation, i.e., the set of resources that a thread needs when in that state.

The ECharts notation [9] holds promise as a means for specifying a SIP servlet's synchronization states and transitions. An ECharts machine is essentially a state-machine representation of a program unit. Smith *et al.* [24] show how to simplify the development of SIP servlets using ECharts by creating an adaptation layer between ECharts and the SIP Servlet API. By augmenting the notation to associate sets of resource names with states, we should be able to generate the negotiator object that a given EChart machine would use. Moreover, we could generate code within the EChart machine so that it will notify the generated negotiator of a change in synchronization state.

Figure 7 depicts an elided example of an EChart machine specification, extended with optional `sync-constraint` clauses that specify the resource needs associated with a given state. The specification declares three states, `IDLE`, `MERGING`, and `FINALIZING`, all of which declare synchronization constraints. The declaration associated with `MERGING` specifies that when the machine is in this state, it requires exclusive access to the resources bound to the resource names `source` and `target`. By contrast, when the machine is in either of the states `IDLE` or `FINALIZING`, it does not require exclusive access to any resources. The resource names `source` and `target` are bound to values in the action on the transition from synchronization state `IDLE` to state `MERGING`. These values are encoded as part of the invite message, which is passed as a parameter to the constructor (not shown in the figure) of this ECharts machine.

6 Conclusions and future work

With respect to synchronization concerns, current container architectures are lacking, both in terms of precisely documenting the policies used to synchronize threads and in the configurability of the policies themselves. Using examples from the telephony domain, we illustrated how these deficiencies manifest in servers that are prone to deadlock in use cases that occur in practice. Interestingly,

```
public machine CallMergeFSM {
  <*
    FeatureBox box;
    SipPort source, target;
    BoxPort boxPort;
  *>

  initial state IDLE sync-constraint { }

  state MERGING sync-constraint { source.SAS, target.SAS }

  state FINALIZING sync-constraint { }

  transition IDLE - boxPort ? Invite / {
    source = message.getAttribute("source");
    target = message.getAttribute("target");
    } -> MERGING;
  ...
```

Fig. 7. ECharts representation of the elided CMG machine, extended with contracts

these deadlocks are avoidable in principle, but not under the implicit-locking policy adopted by some vendors in the absence of precise guidance by the JSR 116 standard. The examples are symptomatic of the problems that arise when a standard is silent on an important non-functional concern and vendors are left to fill in the gaps.

To address these deficiencies, the research community must investigate means to precisely specify and document existing policies and to develop new, more configurable policies that account for more fine-grain needs of a given service. Clearly, the next version of the SIP Servlet API should provide some means for explicit synchronization with deadlock avoidance. Our proposal, inspired by the concurrency controller pattern involves a minimum of new mechanism at the API level, while allowing the generation of powerful synchronization protocols from highly declarative specifications.

Acknowledgements: Partial support was provided for this research by NSF grant CCF 0702667, LogicBlox Inc., and AT&T Research Laboratory. The authors also thank G. Bond, E. Cheung, H. Purdy, T. Smith, V. Subramonian, and P. Zave for indispensable explanations, feedback and guidance.

References

1. BEA white paper: BEA WebLogic Server 10—the rock-solid foundation for SOA, 2007.
2. BEA WebLogic SIP Server - developing applications with WebLogic SIP Server, Dec. 2006.

3. R. Behrends. *Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages*. PhD thesis, Michigan State University, East Lansing, Michigan USA, Dec. 2003.
4. R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of FSE'2000*, 2000.
5. R. Behrends, R. E. K. Stirewalt, and L. K. Dillon. A self-organizing component model for the design of safe multi-threaded applications. In *Proc. of the ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'05)*, 2005.
6. A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. of the IEEE International Conference on Automated Software Enginerring*, 2004.
7. M. R. Blaha and W. J. Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.
8. G. Bond et al. Experience with component-based development of a telecommunication service. In *Proc. of the Eighth ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'05)*, 2005.
9. G. W. Bond and H. Goguen. ECharts: Balancing design and implementation. In *In Proceedings of the* 6^{th} *IASTED International Conference on Software Engineering and Applications*, pages 149–155. ACTA Press, 2002.
10. X. Deng et al. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. of the IEEE International Conference on Software Engineering (ICSE'02)*, 2002.
11. S. D. Fleming et al. Separating synchronization concerns with frameworks and generative programming. Technical Report MSU-CSE-06-34, Michigan State University, East Lansing, Michigan, 2006.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.
13. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA'2003)*, 2003.
14. M. Herlihy et al. Software transactional memory for dynamic-sized data structures. In *Proc. of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003.
15. A. Kristensen. JSR 110: SIP Servlet API version 1.0, Feb. 2003.
16. C. Lopes. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.
17. V. Luchangco and V. J. Marathe. Transaction synchronizers. In *2005 Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL'05)*, Oct. 2005. held at OOPSLA'05.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
19. J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison–Wesley, second edition, 1985.
20. J. Rosenberg et al. SIP: Session Initiation Protocol, 2002. RFC 3261.
21. D. J. Rosenkrantz, R. E. Stearns, and I. Philip M. Lew is. System level concurrency control for distributed database syst ems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.
22. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison–Wesley, second edition, 2004.

23. Sailfin: https://sailfin.dev.java.net.
24. T. M. Smith and G. W. Bond. ECharts for SIP servlets: a state-machine programming environment for VoIP applications. In *IPTComm '07: Proceedings of the* 1st *international conference on Principles, systems and applications of IP telecommunications*, pages 89–98, New York, NY, USA, 2007. ACM.
25. R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared memory sytems. In *Proc. of the* 29th *Annual IEEE/NASA Software Engineering Workshop*, 2005.
26. Apache Tomcat: http://tomcat.apache.org.
27. J. Wilkiewicz and M. Kulkarni. JSR 289 PR: SIP Servlet Specification v1.1.