Contract-Based Synchronization of IP Telecommunication Services: A Case Study

Yi Huang
Michigan State University
East Lansing, MI 48824, USA
huangyi7@cse.msu.edu

Laura K. Dillon
Michigan State University
East Lansing, MI 48824, USA
ldillon@cse.msu.edu

R. E. K. Stirewalt
LogicBlox, Inc.
Atlanta, GA 30309, USA
kurt.stirewalt@logicblox.com

ABSTRACT

Communication middleware, like J2EE and OCCAS, facilitates development and deployment of IP telecommunication services by automating various cross-cutting concerns, such as those related to messaging and security. This middleware is highly concurrent, with threads executing methods that an application programmer writes to carry out the service logic. While the middleware manages lifecycle concerns of threads, the problem of synchronizing them is left to the application programmer. Unfortunately, this synchronization code can be complex and prone to error. Moreover, it can easily obscure the service logic. Our prior work proposed solving these problems using a middleware framework to automatically synchronize concurrent service executions based on declarative synchronization contracts. This paper describes an implementation of our synchronization framework and a case study using it. The case study demonstrates the extent to which contract-based synchronization facilitates refinement of a finite-state design to code and improves design transparency. It also examines the impact on performance of the subject application. A conclusion of the case study is that contract-based synchronization could provide a foundation for automatic generation of IPT services from finite-state designs.

1. INTRODUCTION

Modern IP telecommunication (IPT) services are difficult to develop. This difficulty can be attributed to a variety of reasons. First, communication is typically asynchronous. A service must react to any legal message sent by any of a number of autonomous subscribers at any time. A message may initiate a new *service execution* on behalf of the sender or it may affect a service execution already in progress. In reacting to a message, a service may also create and send messages to other subscribers and sub-services.

Second, a service must manage dynamically changing numbers of service executions on behalf of different subscribers. To this end, it must keep track of the progress made by each service execution—in other words, it must maintain execution states. When a service receives a message pertaining to a particular service execution, it must determine how to process the message based on the service execution's current state and then advance its state accordingly. Maintaining such state information is complex, particularly when the states of multiple service executions are used and updated simultaneously.

Third, current standards in this domain do not address synchronization. *Session Initiation Protocol* (SIP) is the *de facto* communication protocol, and JSR 289 [22] is the industry standard for building SIP-based services. According to this standard, a service is implemented as one or more SIP *servlets*[1] and deployed to a communication middleware, called a SIP servlets *container*. The container automates details of SIP and manages runtime resources, thereby separating theses concerns from the servlets that implement the service logic. However, the obligation of synchronizing concurrent service executions has proven difficult to separate. Because issues raised by concurrency in this domain are complex and not fully understood, JSR 289 does not mandate any particular synchronization model [22, Section 6.4.2]. To prevent data races and yet allow concurrent processing of non-interfering messages, an application programmer must therefore write code to acquire and release locks on shared resources. Such low-level locking is prone to subtle synchronization errors, which are notoriously difficult to diagnose and correct.

In previous work, we elaborated several key dimensions of the thread synchronization problem for IPT services and proposed a solution based on *synchronization contracts* [13, 14]. This solution leverages a common convention in the IPT domain, which copes with the complexity of a service by representing the service's business logic as a state-machine and by emulating this state-machine in the servlet [18, 24]. Specifically, a synchronization contract encodes the control logic of a *business machine* and, for each transition of this machine, declares the shared resources accessed by the code that implements the transition. This manner of specification makes it possible to encapsulate the synchronization code in a middleware layer. Moreover, this layer raises the level of abstraction for programming services, facilitating refinement of a business-machine design into code.

This paper describes an implementation of this framework and a case study meant to assess if a contract-based synchronization framework provides a suitable foundation for automated generation of IPT services from finite-state designs. For the synchronization framework, we re-engineered the *Synchronization Negotiation Framework for SIP Servlets (SNeF4SS)* [13], so that it provides an API suited to abstract business machines. The case study focused

[1]and, possibly, HTTP servlets, supporting classes, and other resources (e.g., JSP pages and images); our current framework does not handle HTTP servlets, but we do not anticipate any new obstacles to doing so.

on two main questions. First, we sought to determine if encapsulating the synchronization code within SNeF4SS would facilitate transparent refinement of a realistically complex business-machine design into code. For the subject application, we compose a multi-party dating service running on one server with a media service running on a different server. Such service composition complicates the synchronization logic due to the need to maintain consistent state information across multiple concurrent service executions [18]. A second goal for the study was to learn how use of the framework would affect performance of this application. IPT services have stringent quality of service (QoS) requirements [8, 17, 21]. We therefore wanted to determine how dynamically evaluating and enforcing synchronization contracts would impact performance of this application.

The remainder of the paper is organized as follows. Section 2 provides background on design, development, and deployment of IPT services and on closely related work. It also presents a business-machine design of the dating service for purposes of illustration and for use in later sections. Section 3 then describes SNeF4SS. The next two sections present the case study. Specifically, Section 4 describes the implementations developed for the study and considers software engineering tradeoffs, and Section 5 describes and discusses the performance evaluation. We present concluding remarks in Section 6.

2. BACKGROUND

This section provides background needed to understand the case study and discusses some closely related work. We first provide a brief overview of the platform upon which modern IPT services are built (Sec. 2.1). We then present the business-machine design used in our case study (Sec. 2.2) and discuss related work on synchronization contracts (Sec. 2.3).

2.1 IPT services and SIP servlet containers

An IPT service implements a *dynamic collaboration* among multiple *endpoints*, e.g., SIP devices and other IPT services. Each endpoint can be viewed as playing a *role* in the collaboration, where the role determines the endpoint's obligations and relationships to other endpoints, much like the role played by an object determines that object's obligations and relationships to other objects in an OO collaboration [20]. For example, the roles of the endpoints in a teleconferencing service might include "administrator," whose player is responsible for maintaining the service; "leader," whose player started and is therefore responsible for ending a specific teleconference; and any number of "participants," each of whose players has joined the call. Communication with endpoints is accomplished by exchanging messages asynchronously over two-way signaling channels.

An IPT service is implemented as one or more servlets, which are deployed to a SIP servlets container. The container simplifies programming by automating routine details of SIP and other generic concerns, like resource management and security. Each servlet is a stateless object to which the container dispatches incoming SIP messages for processing. Being stateless, a servlet may host multiple threads. When processing a message, a servlet stores information it will need in order to process subsequent messages in programmer-defined *attributes* of container-managed objects, called *sessions*. A container automatically associates two types of sessions with a SIP message: an *application session* and a *SIP session*. An application session typically stores application-specific information, such as the identity of the service "administrator." Different executions of a given service often share a single application session to be able to exchange data. In contrast, they typically aggregate disjoint sets of SIP sessions. Each SIP session objectifies a signaling channel between the service execution and one of its role players (endpoints). The SIP session stores information needed to send messages to the role player, e.g., the role player's media characteristics, as well as information needed to carry out the business logic, e.g., the role played by the endpoint and the sessions needed to signal other role players. We refer to the service execution that creates a SIP session as the role player's *parent execution*.

To ensure responsiveness, a container executes in a dedicated thread, listening for incoming messages on the network and for outgoing messages generated by the services it hosts. Upon receiving a message bound for one of these services, the container determines the servlet to route the message to and retrieves the message's application session and SIP session, or creates them if they do not exist. It then dispatches a SIP thread to process the message, passing that thread the message and the associated servlet and sessions. The thread processes the message by invoking a *message handler*, which the application programmer overrides to implement the service's business logic. For instance, the application programmer typically implements the logic for processing an `INVITE` message in the `doInvite` method, the logic for processing an `OK` message in the `doSuccessResponse` method, and so on. Collectively, these methods are referred to as `doXXX` methods.

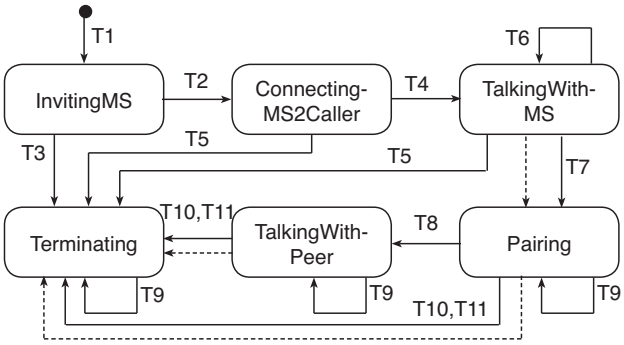
Smith and Bond argue that a major source of the complexity of modern telecommunication services stems from the need to maintain substantial execution state, and that use of the Echarts state-machine programming notation can alleviate much of this complexity [18]. The SIP-container API provides only low-level operations for maintaining the state of a service execution, which must be implemented using attributes in the sessions associated with the service execution. Specifically, the API provides operations to create and destroy SIP sessions and to save and retrieve values of session attributes. It also provides operations for mapping sessions to session identifiers (IDs), suitable for transmitting over a network, and for retrieving sessions from their IDs. Echarts permits an alternate approach. The state maintenance logic is written in Echarts and then automatically translated into a SIP servlets application.

As IPT applications have evolved from single-service routing applications to applications composed of multiple services and involving multiple autonomous endpoints, the need to synchronize concurrent service executions without incurring deadlock has emerged as another significant issue for these applications. Some container implementations enforce synchronization policies automatically and also provide proprietary synchronization commands. For example, the container implemented by one popular vendor automatically locks the application session before processing any message bound to a service. A proprietary command also permits grouping sequences of operations into special transactions. Fundamental problems with these and other such mechanisms are discussed in [14]. SNeF4SS (Section 3) leverages synchronization contracts to allow developers to program SIP services using state-machine designs in the knowledge that the executions will synchronize correctly.

2.2 State-machine design of a dating service

The subject of our case study is a multi-party dating service, which sets up a phone call between pairs of subscribers who dial into the service to meet someone new. [2] The dating service delegates responsibility to collect match preferences to a media service, which subsequently messages the dating service with the identities

[2]Much like `omegle.com` and `chatroulette.com`.



Annotations (label: guard -> actions)
T1: (def Caller)?INVITE -> (def Msr)!INVITE(Caller)
T2: Msr?OK -> Caller!OK
T3: Caller?CANCEL -> Caller!OK; Msr!CANCEL
T4: Caller?ACK -> Msr!ACK
T5: Caller?BYE -> Caller!OK; Msr!BYE
T6: Msr?MESSAGE(def PeerToTry) && (PeerToTry == null || not (peerToTry in TalkingWithMS)) -> Msr!OK; Msr!MESSAGE(PeerToTry)
T7: Msr?MESSAGE(def PeerToTry) && PeerToTry != null && PeerToTry in TalkingWithMS -> Msr!OK; def Peer = PeerToTry; Caller!INVITE(Peer); Peer to Pairing; Peer!INVITE(Caller); Msr!BYE; Peer.Msr!BYE
T8: Caller?OK -> Caller!ACK
T9: Msr?MESSAGE(_) -> Msr!OK
T10: Msr?ERROR -> Caller!ERROR
T11: Caller?BYE -> Caller!OK; Msr!BYE; Peer!BYE, Peer to Terminating

Figure 1: Business-machine design for a dial-up dating service

of callers who match each others' preferences. This media service is deployed as a separate SIP service.[3]

Figure 1 depicts a business-machine design describing the behavior of one execution of this service. Informally, a state (rounded rectangle) records the service execution's progress between messages and a transition (labelled arrow) denotes the processing of a message. To reduce clutter, we depict transitions with identical source states and identical target states by annotating a single arrow with multiple labels. The Annotations Table associates each transition with a *guarded command*, of the form *guard* `->` *action_list*, which indicates the message that triggers the transition and the effects of processing this message.

The guarded command notation combines ideas from CSP [12] and collaboration-based design [23]. A *receive event*, denoted `Role?msg`, models the reception of a message of type `msg` sent by the endpoint playing role `Role`. A receive event is *enabled* if the message to be processed is of the indicated type and was sent by the designated endpoint's role player. Because SIP messaging is asynchronous, the sending of a message is modeled by an action rather than an event. Denoted `Role!msg`, the *send action* signifies that a message of type `msg` is sent to the endpoint playing role `Role`. Business data transmitted in a message, if any, are represented by arguments in a send action and by formal parameters in a receive event. Message transmission is asynchronous and messages need not arrive in the order sent. A role designates an endpoint, e.g., `Caller` refers to the phone that dialed the dating service. The qualifier `def` preceding a role signifies that the event/action defines the role (i.e., binds it to an endpoint).

For example, when Mary dials the dating service, her phone sends an `INVITE` message to the servlet container that hosts the dating service. Upon receiving this message, the container dispatches it to a SIP thread for processing, thereby initializing a new execution of the dating service, which we refer to as Mary's service execution. The initial transition (`T1`) represents the processing of this message. It binds the `Caller` role in Mary's service execution to Mary's phone (the endpoint that sent the message). It then creates and sends a new `INVITE` message to a media service, binding the media-service execution created by the media service when it receives this message to the `Msr` role in Mary's service execution. This latter message carries an identifier designating Mary's phone (the caller), which the media service will need if Mary matches another caller's preferences. When the SIP thread processing Mary's original `INVITE` message terminates, Mary's service execution is in state `InvitingMS`. Because the SIP sessions for the endpoints that play the `Caller` and `Msr` roles are created in processing this message, Mary's service execution is the parent service execution for both endpoints.

If the media-service execution subsequently responds with an `OK` message, Mary's dating-service execution relays this response to her phone (`T2`) and transitions into state `ConnectingMS2Caller`. A subsequent `ACK` response from Mary's phone is similarly relayed to the media service (`T4`). This exchange completes a three-way handshake between Mary's phone and the media service, permitting Mary's phone and the media service to exchange media directly with one another. While Mary's phone and the media service are connected, Mary's service execution is in state `TalkingWith-MS`.

Normally, Mary's service execution waits in this latter state until it either receives a message containing the identity of a caller that matches Mary's preferences or Mary is selected by some other caller. Either of these events requires that Mary's service execution performs a transition *synchronously* with another caller's service execution. For instance, suppose Mary signals a preference for Tom, the caller in another dating-service execution. Her media-service execution will then notify her dating-service execution of her selection. However, by the time Mary's service execution receives this notification message, Tom might not be available (e.g., Tom may have hung up or selected yet a third caller to be connected with). Thus, to ensure that neither service execution gets into an inconsistent state, the reception of the notification message should trigger a simultaneous state change in *both* Mary's and Tom's service executions, but only if both are still available to be paired.

This rendezvous style of synchronization can be implemented by acquiring locks on all sessions used to record the execution state of

[3]For an early proof-of-concept study, we used a design of the dating service that abstracted the process whereby a caller selects a matching caller by a function that simply returns a reference to an arbitrary caller [13]. However, this simplification is unrealistic and, in eliminating a crucial autonomous role and thus a major source of asynchrony, also significantly reduces the complexity of both the service logic and the required synchronization. The subject of the study in this paper is a major redesign the original dating service.

any *peer* in the rendezvous, and then modifying the attributes of these sessions in a single critical section. In a business machine, we show these rendezvous operations more abstractly using guard conditions and actions that refer to the states of multiple service executions and using special transitions with no guarded commands. Specifically, the guard condition (*role* `in` *state*) queries the state of the parent service execution of the endpoint designated by *role*, returning true if the service execution is in state *state* and false otherwise; and the action (*role* `to` *state*) transitions this same service execution to state *state*. A transition with no guarded command (dashed arrow) signifies that when a service execution is in the transition's source state and not processing any message, another service execution may perform an action that leaves it in the transition's target state.

Continuing with our example, suppose Mary's service execution receives a `MESSAGE` message while it is still in state `TalkingWithMS` and the identifier passed in for `PeerToTry` designates the caller in Tom's service execution. If Tom's service execution is also in state `TalkingWithMS`, Mary's service execution transitions both itself and Tom's service execution into state `Pairing` (`T7`). In `Pairing`, Mary's service execution attempts to connect Mary's phone and the peer's phone (`T8`). On the other hand, if Tom's service execution is not in state `TalkingWithMS` when Mary's service execution receives the `MESSAGE` message, Tom is no longer available, and Mary's service execution arranges for Mary to select a different caller (`T6`). At any time, Mary can terminate her service execution by hanging up (`T3`, `T5`, or `T11`). Additionally, while the dating service is trying to connect the media service to Mary's phone, the media service can signal an error has occurred (`T10`).

For our case study (Section 4), we refined the business-machine design in Figure 1 into three implementations—two that run directly on the container API and one that runs on our middleware API. We then compared the alternative implementations with respect to design transparency and performance.

2.3 Related work on synchronization contracts

Before describing our framework, we briefly discuss two existing contract-based synchronization models, SCOOP [16] and Szumo [1], which have influenced our work. Both are designed for general OO languages, in which communication is primarily synchronous. To use SCOOP for controlling access to sessions, sessions should be implemented as *separate objects* [16]. But it is difficult to see how a message handler would be programmed to use SCOOP contracts (method preconditions) to emulate transitions of a business machine, as the handler must infer the sessions (separate objects) it needs to access—their identities are not known when the handler is invoked.

The contracts used with SNeF4SS are modified Szumo-style contracts. In Szumo, a synchronization contract is declared in a class interface for use with all objects of that type (instances of the class). The synchronization contract for an object o distinguishes a finite set of *synchronization states* of o and associates a finite set of *handles* with each synchronization state, where handles are local expressions that reference shared objects. The contract asserts that o needs exclusive access to all objects referenced by handles that the contract associates with a synchronization state s from the time that o enters s until it next leaves s. In SNeF4SS, a synchronization contract instead declares, for each transition in a business machine, a guard condition and a set of handles, which are to be used in determining, respectively, when the transition is taken and the sessions it will access. We do not associate handles with states of a business machine because a state in a business machine represents a period of inactivity; shared data needs to be protected when a service execution processes a message, not when it is idle. Additionally, because a servlet has no state, a programmer does not specify handles using local expressions, as in Szumo; instead, she specifies how to navigate the message-processing context to locate the sessions that a transition will manipulate.

A thread in an OO program has more in common with a service execution than with a SIP thread. A thread in a SIP container processes just a single message and is very short-lived. Because acquiring locks in the midst of processing a message would incur unpredictable delay, and might unduly tie up data needed by other service executions or even cause deadlock, doing so is not advised. For these reasons, SNeF4SS ensures that transitions are transactional [2]. Because many actions performed when processing a message cannot be retracted, it uses a pessimistic locking protocol, acquiring locks for all needed sessions prior to performing any of a transition's actions.

3. SNeF4SS

SNeF4SS is a middleware layer between the servlet and the container. Its purpose is to encapsulate the implementation of a servlet's synchronization logic. Instead of embedding operational synchronization code into message handlers, the application programmer declares the conditions under which a servlet performs transitions and the sessions that it accesses when performing them (Sec. 3.2). These declarations are the synchronization contracts. She also writes a message handler, which implements the actions that are performed on transitions (Sec. 3.3). Consulting the contract and the message handler, the framework automatically emulates the business machine while preventing data races and deadlock (Sec. 3.4). To facilitate the description of SNeF4SS, we begin by describing the pseudocode conventions used throughout the remainder of the paper (Sec. 3.1).

3.1 Pseudocode conventions

Prior to writing the contracts or any message handlers, the programmer must decide how she will represent concepts in the business machine as data objects that a message handler can manipulate when processing a message. Because a servlet is stateless, the handler must use the servlets-container API to navigate from the message-processing context to any data it needs. Thus, the data (or references to the data) are typically stored as values of attributes in SIP sessions.

For the dating service, we give every SIP session an attribute, named `role`, that is assigned one of two values: `"Caller"`, if the SIP session objectifies the service execution's channel to its caller; or `"Msr"`, if it objectifies the service execution's channel to the media service. Before processing a message, the handler checks the value of this attribute to determine which endpoint sent the message. The other attributes in a SIP session depend on the value of the `role` attribute:

- if `role` == `"Caller"`:
  `cs`: the state of the service execution
  `msr`: the SIP session for the service execution's channel to the media service
  `p2t`: the SIP session in a peer's service execution for the channel to the peer execution's caller
- if `role` == `"Msr"`:
  `caller`: the SIP session for the service execution's channel to its caller

We use this attribute scheme and a simple version of *navigation expressions* [9] in describing both contracts and the processing

```
nav_expr   =  mess_desc
           |  sess_desc, { "." , name }*
mess_desc  =  "msg", ".", ( "method" | "payl" )
sess_desc  =  "msg", ".", ( "apps" | "sips" )
           |  name
```

Figure 2: SNeF4SS navigation expressions. (Extended BNF notation: "typewriter" – terminal; *italics* – non-terminal; comma – concatenation; vertical bar – alternation; set brackets – (optional) repetition; equal – definition; parentheses – grouping.)

performed in message handlers. A navigation expression is a "dot-separated" sequence of two or more names (Fig. 2). It consists either of a message descriptor or of a session descriptor, where the latter may be followed by any number of names. A message descriptor stands for information that is found in the message. We use two in this example: `msg.payl` stands for the information transmitted in the payload of a message, and `msg.method` stands for the method named in the first line of a SIP request message or the type of response designated by the response code in the first line of a SIP response message. Thus, for example, when processing a response message, the navigation expression `msg.method` signifies the response type (e.g., `OK`). In contrast, a navigation expression that contains a session descriptor specifies how to locate session data. The session descriptors `msg.apps` and `msg.sips` indicate that the traversal starts in the message's application session and in the message's SIP session, respectively. Any other name used as a session descriptor denotes a variable that references a session. Additional names attached to a session descriptor designate a series of attributes to be traversed to reach a data object. For example, when processing a message sent by a service execution's caller, the navigation expression `msg.sips.msr` stands for the SIP session for the service execution's channel to the media service.

Finally, we introduce three abstract functions to use in pseudocode descriptions.

- `send(method, payl, chan)`: Creates and sends a message. Arguments indicate the method or response code, the payload, and the channel (recipient).
- `getID(ss)`: Returns the session identifier that the container associates with a session.
- `getSession(ssid)`: Returns the session indicated by a session ID.

3.2 Synchronization contracts

SNeF4SS synchronization contracts declare guard conditions for the transitions in a business machine and handles for the sessions that the transitions manipulate. These declarations permit the middleware to automatically acquire sessions, as they are needed, in order to emulate execution of the business machine without incurring data races or deadlock.

SNeF4SS reads an XML encoding of a table defining a servlet's contracts. Table 1 depicts some of the contracts for the dating service example. Each row of this table names a transition (first column), specifies a guard expression (middle column) and a set of handles (last column), and defines any variables used in the contract (spanning the last two columns). The guard condition specifies when the transition is executed as a boolean expression. The handles specify the set of data objects to which a message handler assumes it has exclusive access for the duration of a transition—namely, all sessions that are traversed in evaluating any of the handles. For example, the first row of Table 1 specifies that an `OK` response, if sent by the media service and received while a service execution is in state `InvitingMS`, triggers transition `T2`, and that during execution of this transition, the handler assumes it has exclusive access to the SIP session for the caller (`msg.sips.caller`) and, because the SIP session for the media service (`msg.sips`) is traversed in evaluating the handle, to this latter SIP session as well. The table illustrates contracts for the transitions triggered by two types of SIP messages: `OK` messages and `MESSAGE` messages. The former is representative of a response type and the latter is the request type requiring the most complex processing in this example. SNeF4SS automatically translates navigation expressions into appropriate calls on the container API when evaluating contracts.

3.3 The message handler

When writing a servlet that will run directly on the container API, a programmer typically implements the actions described by a business machine in multiple message-specific `doXXX` methods. For a servlet that will use the SNeF4SS API, the programmer instead overrides a single abstract handler method, called `doTransition` (Fig. 3).

The `doTransition` method is passed the name of the transition to emulate and the message to process. Based on the transition name, it selects a branch of a conditional statement, which carries out the actions for that transition. For example, the first branch of the conditional (lines 2–7) in Figure 3 implements the actions performed by the initial transition, `T1`, of the business machine in Figure 1. Briefly, it initializes attributes `role` and `cs` in the message's SIP session (lines 2 and 3, resp.); creates a new SIP session to serve as the service execution's channel to the media service and assigns this new session to attribute `msr` (line 4); initializes attributes `caller` and `role` in this new session (lines 5 and 6, resp.); and sends an `INVITE` message to the media service (line 7) containing an identifier, which the container associates with the caller's SIP session.

When refining a business machine into an actual servlet, a programmer substitutes calls to the container API for the pseudocode shown in our figures. This step is routine, but tedious and prone to error because of the low level at which the programming must currently be done. Understanding the actual servlet code requires a deeper understanding of SIP than can be conveyed here. However, the actual source code is available for download at our website.[4] Moreover, a conclusion of our case study is that this translation could be largely automated.

3.4 Operation of the framework

When a SNeF4SS servlet is deployed to a container, the servlet's contract is loaded and parsed into an internal representation for later evaluation relative to different message-processing contexts. Subsequently, upon being dispatched, a SNeF4SS thread evaluates this internal representation relative to the message that it was dispatched to process, producing a *dynamic contract*. The dynamic contract contains the name of the transition, if any, that the message triggers, together with references to the sessions obtained by evaluating the handles associated with this transition. If the dynamic contract is null, the message does not trigger any transition, and so the SNeF4SS thread just terminates without calling the handler. This helps to automatically absorb resent or irrelevant messages. More typically, the dynamic contract is non-null

[4] http://www.cse.msu.edu/sens/szumo/SNeF/

Name	Guard	Handles
T2	`msg.method == OK && msg.sips.role == "Msr" && msg.sips.caller.cs == "InvitingMS"`	`msg.sips.caller`
T6	`p2t = getSession(ssid = msg.payl)`	
	`msg.method == MESSAGE && msg.sips.role == "Msr" && msg.sips.caller.cs == "TalkingWithMS" && (p2t == null || p2t.cs != "TalkingWithMS")`	`msg.sips.caller, p2t`
T7	`p2t = getSession(ssid = msg.payl)`	
	`msg.method == MESSAGE && msg.sips.role == "Msr" && msg.sips.caller.cs == "TalkingWithMS" && p2t != null && p2t.cs == "TalkingWithMS"`	`msg.sips.caller, p2t.msr`
T8	`msg.method == OK && msg.sips.role == "Caller" && msg.sips.cs == "Pairing"`	`msg.sips`
T9	`msg.method == MESSAGE && msg.sips.role == "Msr" && (msg.sips.caller.cs == "Pairing" || msg.sips.caller.cs == "TalkingWithPeer" || msg.sips.caller.cs == "Terminating")`	`msg.sips`

Table 1: Synchronization contracts for transitions triggered by `OK` and `MESSAGE` messages.

and the thread then performs a negotiation protocol to acquire the needed sessions. During negotiation the thread may yield to other threads contending for some of the same sessions in order to prevent deadlock. When it succeeds in acquiring the needed sessions, a SNeF4SS thread invokes the servlet's `doTransition` method, passing it the name of the transition and the original message. Finally, when the call to `doTransition` returns, the thread releases all sessions and terminates.

In essence, SNeF4SS encapsulates a distributed concurrency controller[3]. It automatically creates an isolated zone [7, Chap. 5] containing the resources needed by the handler. There are many protocols for acquiring a set of resources without incurring deadlock [25]. Known protocols range in complexity, from relatively simple algorithms [10], which may be overly pessimistic or unfair, to complex algorithms [1, 6], which can permit more concurrency and be fairer, but at some extra costs in overhead. To separate the implementation of the negotiation protocol from the rest of the framework, SNeF4SS encapsulates the protocol implementation in *negotiator* objects, which implement a generic Negotiator API.

Encapsulating the synchronization code within a framework in this manner promises a number of benefits. First, it frees the application programmer to focus on design and implementation of the business logic. As illustrated in Figure 3, the servlet code does not tangle business code with synchronization code. The business logic is thus relatively easy to trace to the servlet code. Second, experts in concurrent programming can write the implementation of the locking protocol to be efficient under expected usage profiles and validate that it avoids deadlock and starvation. The application programmer, who is an expert in the business domain, but may not be proficient in thread programming, does not write the code to acquire and release sessions. Use of the framework can thus offer strong guarantees with regard to efficiency and absence of concurrency errors. Third, alternative synchronization protocol implementations can be seamlessly swapped into the framework to accommodate changes in a service's usage profile. Concurrency experts can develop custom negotiators by programming to the SNeF4SS Negotiator API [15]. The current SNeF4SS toolkit provides three alternative negotiators for an application programmer to use off-the-shelf.

4. CASE STUDY

Having introduced the business-machine design for the dating service and described our framework, we now turn to describing the case study. We developed and compared three implementations of the design described in Section 2.2 in an attempt to understand tradeoffs between design transparency and performance. The goal was to determine if additional investment in generative programming of IPT services from high-level designs and synchronization contracts is warranted. Toward this end, the case study looked at two key questions:

1. How does servlet code written to run directly on the container API compare to servlet code written to run on SNeF4SS with respect to design transparency and code complexity?

2. What are the performance costs to use contract-based synchronization instead of hand-written synchronization?

This section considers Question 1, which is difficult to quantify, but is important for reasoning about correctness and determining how well the framework would support automated generation of IPT services. The next section considers Question 2, which is a concern because quality-of-service is a critical differentiator among providers in this domain [5].

We originally intended to re-engineer an existing service to use as the subject of the case study. However, we abandoned this plan for two reasons. First, the field of IPT services has grown out of the telecommunications domain, in which software tends to be proprietary. The only open source services that we found were those packaged with open source containers to show how to use the container API; they were far too simple to serve as subjects for our case study. We are currently working with the SPEC SIP Committee on development of a benchmark for evaluating SIP application servers,[5] but this effort started long after we performed the case study reported here. A second reason for developing a service from scratch was to be able to address the two questions identified. For Question 1, we needed to start with a well-engineered business-machine design. For Question 2, we needed to control carefully for both differences in the implementations and for factors not related to synchronization that could affect our measurements. However, mindful of this potential threat to validity, we provide the full

[5]http://www.spec.org/specsip

```
  void doTransition (name:String, msg:SIPMessage){
1   if (name == "T1") { caller = msg.sips
2     caller.role = "Caller"
3     caller.cs = "InvitingMS"
4     caller.msr = new SIP session
5     caller.msr.caller = caller
6     caller.msr.role = "Msr"
7     send(method = INVITE,
           payl = getID(ss = caller),
           chan = caller.msr) }
    //
8   else if (name == "T2") { msr = msg.sips
9     send(method = OK, chan = msr.caller)
10    msr.caller.cs = "ConnectingMS2Caller" }
    //
    ...
11  else if (name == "T6") { msr = msg.sips
12    send(method = OK, chan = msr)
13    send(method = MESSAGE, payl = msg.payl,
           chan = msr) }
    //
14  else if (name == "T7") { msr = msg.sips
15    caller = msr.caller
16    p2t = getSession(ssid = msg.payl)
17    send(method = OK, chan = msr)
18    caller.p2t = p2t
19    caller.cs = "Pairing"
20    p2t.cs = "Pairing"
      // connect caller and p2t
21    send(method = INVITE, chan = caller,
           payl = p2t.SDPinfo)
22    send(method = INVITE, chan = p2t,
           payl = caller.SDPinfo )
      // tear down both channels to media service
23    send(method = BYE, chan = msr)
24    send(method = BYE,
           chan = p2t.msr) }
    //
25  else if (name == "T8") {caller = msg.sips
26    send(message = ACK, chan = caller
27    caller.cs = "TalkingWithPeer" }
28  else if (name == "T9") {
29    send(message = OK, chan = msg.sips) }
    ...  }
```

Figure 3: Pseudocode for part of a handler to use with SNeF4SS

source code for all versions of the dating service developed for this study and instructions for replicating our results.[6]

For the case study, we implemented the business machine from Section 2.2 three times. Two of the implementations target the container API and the third targets the SNeF4SS API. We refer to the first two as *manual versions*, because they embed custom code to synchronize concurrent executions, and to the third as the *contract version*, because it leverages synchronization contracts. We then compared the manual versions to the contract version on measures that speak to the two questions.

To facilitate comparison and highlight tradeoffs between design transparency and efficiency in the three implementations, we first describe how the business logic is expressed in servlet code, without concern for synchronization (Section 4.1). Then we describe how the different approaches for implementing the synchronization logic affect this servlet code (Section 4.2).

[6]http://www.cse.msu.edu/sens/szumo/SNeF

4.1 Implementing the business logic

Before writing any code, we designed a scheme for storing and retrieving persistent data in SIP sessions. The requirements for such a scheme are similar, whether targeting the container API or the SNeF4SS API. When processing a message, the handler must find any data it needs by navigating the message-processing context. The scheme that we started with for all versions is that described in Section 3.1.

Manual versions. The manual versions must conform to the programming rules in JSR 289 [22]. According to this standard, the servlet inherits from the class `SipServlet` and overrides the appropriate `doXXX` methods. When a SIP thread is dispatched to process a message, it invokes the servlet's `service` method. This latter method examines the message type and then invokes the appropriate `doXXX` method. A SIP thread performs these operations automatically. Therefore, the application programmer need only write code for the `doXXX` methods to implement the business logic.

Because the `doXXX` method that a SIP thread invokes depends on the type of the message to be processed, we partitioned the transitions of the business machine, putting all transitions that are triggered by a given message type into the same equivalence class. We then implemented the transitions for an equivalence class as a `doXXX` method.

```
  void doSuccessResponse (msg:  SIP Message){
1   if (msg.sips.role == "Caller") {
2     if (msg.sips.cs == "Pairing") {
        //T8 (Fig. 3, lines 25-27)
    }}
3   else if (msg.sips.role == "Msr") {
4     if (msg.sips.caller.cs == "InvitingMS") {
        //T2 (Fig. 3, lines 8-10)
    }}
  }

  void doMessage(msg:  SIP Message) {
5   msr = msg.sips
6   caller = msr.caller
7   p2t = getSession(ssid = msg.payl)
8   if (caller.cs == "TalkingWithMS") {
9     if (p2t == null or
          p2t.cs != "TalkingWithMS") {
        //T6 (Fig. 3, lines 11-13)
      }
10    else {
        //T7 (Fig. 3, lines 14-24)
    }}
11  else {
      // caller.cs is one of:  "Pairing" or
      //  "TalkingWithPeer" or "Terminating"
      //T9 (Fig. 3, line 28, line 29)
    }
  }
```

Figure 4: Pseudocode for two SIP handlers

To illustrate, Figure 4 shows pseudocode for two representative handlers, `doSuccessResponse` and `doMessage`. Briefly, when the dating service receives an `OK` response message, it checks whether the response was sent by the caller (line 1) or the media service (line 3), and then checks the current state. If the caller sent the response and the service execution is in state `Pairing` (line 2), the handler implements transition `T8`. If the media service sent the response and the service execution is in state `InvitingMS` (line 4),

the handler implements transition T2. In all other cases, the message has no affect on the service execution. The most complex handler is doMessage (lines 5–11). It does not check the role of the sender because the caller does not send MESSAGE messages to the dating service. However, if the service execution is in state TalkingWithMS, the handler checks if the service execution of the prospective peer is also in state TalkingWithMS, implementing transition T7, if it is, and transition T6, otherwise (lines 10 and 9). Finally, the handler executes T9 (a self loop) in all other cases (line 11).

Contract version. For the contract version, instead of inheriting directly from SipServlet, the servlet class inherits from a base class that SNeF4SS defines via subclassing SipServlet. The new base class overrides the service method from the SipServlet class. The SNeF4SS service method does not delegate handling of different messages to different handlers; instead, it uses a negotiator object to learn the transition to emulate and then invokes doTransition, passing both the transition name and the message as arguments. Like a SIP thread, a SNeF4SS thread automatically invokes the servlet's service method. Therefore, the application programmer's job consists of writing code for the doTransition method to implement the actions performed on transitions of the business machine. Section 3.3 describes this process and shows pseudocode for handling OK and MESSAGE messages (Fig. 3).

Discussion. The example pseudocode in Figures 3 and 4 illustrates one advantage of a middleware that leverages contracts to support the business machine abstraction. Declaring transitions and associated guard conditions in separate contracts that the middleware uses to automatically determine the transition to emulate permits the use of a handler whose structure very transparently parallels the structure of the business machine. The application programmer does not need to partition the business logic into handlers for different types of SIP messages, or clutter the handler with code to check the role of the message sender or the execution state. The use of explicit transition names in the SNeF4SS handler makes it self-documenting. Tracing transitions of the business machine to the code in message-specific SIP handlers can be difficult, especially in the absence of comments spelling out the correspondence.

4.2 Implementing synchronization

Threads hosted by the same container must be properly synchronized to protect shared sessions from data races and to prevent deadlock. All three versions of the dating service use the same locking protocol, resource numbering [11], to avoid deadlock. In order to better understand tradeoffs between design transparency, code complexity, and efficiency, one of the manual versions uses the same generic ordering scheme as the contract version—ordering sessions by hash codes that the Sun Java Virtual Machine (JVM) assigns to objects—and the other employs an application-specific ordering scheme.

First manual version. For the first manual version, we tried to retain as much design transparency as possible. Toward this end, we extended the base servlet class with two new methods: acquire( ) aggregates the logic for each message type to infer and acquire locks on all needed sessions; and release( ) releases all locks held by the handler. We then added a call to acquire as the first instruction and a call to release as the last instruction in each doXXX method.[7]

[7]with one exception, doINVITE—In theory, a race on the message's SIP session could occur between doINVITE and doCANCEL, but the likelihood is extremely small and, in any case, the race would be benign because doCANCEL invalidates the signaling channel with the caller.

The application programmer must write the acquire( ) and release( ) methods. It is easy to write a "generic" release( ) method, which simply releases the locks on all sessions held by the message handler. However, the acquire( ) must infer the sessions that should be locked, which requires precisely the information encoded in our contracts. In the absence of explicit declarations encoding this information, the application programmer must write an acquire( ) method that hard codes this information.

To infer the sessions that are needed, the acquire( ) method in our first manual version aggregates the control logic of all handlers: First, it determines the message type, the role played by the endpoint that sent the message, and the state of the service execution. Based on this information, it determines the actions that must be performed and the sessions that will be accessed in performing these actions. During this inference process, the method locks a session when necessary to traverse an attribute in that session (e.g., to infer a session some action will access). Because the traversal order may be inconsistent with the hash-code order used to prevent deadlock, the code to evaluate a navigation expression that traverses multiple sessions without introducing deadlock is complex. It must allow that, to acquire a lock on a new session, the handler may have to release locks on sessions that are already held and reacquire them once it has locked the new session, and that the states of the reacquired sessions may change between the time they are released and the time they are reacquired. To avoid this complexity and extra overhead, in this first manual version of the dating service, we added a cs attribute to the SIP session for each service execution's channel to the media service; this attribute simply duplicates the cs attribute in the SIP session for the service execution's channel to the caller. By duplicating this information in both SIP sessions of a service execution, we arrange that only the SIP session associated with the message is needed in order to determine all sessions that must be acquired.[8] Thus, our acquire( ) method locks the message's SIP session and infers the sessions needed to process the message. It then proceeds to acquire the needed sessions in hash-code order, releasing and reacquiring the message's SIP session if necessary to acquire all the sessions in hash-code order. It is unlikely that, in the time between releasing the message's SIP session and reacquiring it, the cs attribute will have changed. However, if it does, the method can simply restart.

Duplicating the cs attribute in the SIP sessions for both role players in a service execution simplifies the synchronization logic in the acquire( ) method, but requires adding to the servlet of the previous subsection code that keeps the duplicate attribute values consistent. It also means that, at a minimum, a thread must acquire the sessions for both of the role players in a service execution when processing any message. For instance, suppose a thread is dispatched with an OK message sent by the caller in a service execution when the service execution is in state Pairing (transition T8). In the first manual version the thread must acquire both the message's SIP session and the SIP session for the media service in order to update the cs attributes in both sessions; whereas, in the optimized and contract versions, only the message's SIP session is needed.

Second manual version. The second manual version trades design transparency in order to optimize performance. In this version, we embedded synchronization code directly into the branches of handlers, thereby eliminating the need to duplicate the control logic by which a handler infers the actions to perform in the acquire method. Doing the acquisition and release in the same method

[8]for T7, the msr attribute of the peer session can be safely accessed without locking the peer session because the attribute is a "single-assignment" attribute

also allows use of Java synchronized statements instead of explicit locks.

To simplify the locking logic, we developed a custom order for acquiring resources. We first ordered the SIP sessions in any given service execution so that the SIP session for the caller immediately precedes that for the media service. Then we used the hash-code order between the SIP sessions for the callers in different service executions to induce a total order on the full space of SIP sessions. Finally, as a further optimization in this version, we leveraged the knowledge that attribute `role` is a single assignment attribute—i.e., `role` is assigned a value at initialization and is never reassigned—to reduce the size of critical regions.

```
   void doSuccessResponse(msg:SIPMessage) {
1    if (msg.sips.role == "Caller") {
       //a.o.:  msg.sips < msg.sips.msr
2      synchronized (msg.sips) {
3        if (msg.sips.cs == "Pairing") {
           //T8 (Fig. 3, lines 25-27)
     }}}
4    else if (msg.sips.role == "Msr") {
       //a.o.:  msg.sips.caller < msg.sips
5      synchronized (msg.sips.caller) {
6        if (msg.sips.caller.cs ==
             "InvitingMS") {
7          synchronized (msg.sips) {
             //T2 (Fig. 3, lines 8-10)
   }}}}}

   void doMessage(msg:SIPMessage) {
     // (Fig. 4, lines 5 - 7)
8    synchronized (msr) { // factor out Msr!OK
9      // T9 (Fig. 3, line 29)
     }
10   if (p2t == null) {
11     synchronized (msr) {
12       // T6' (Fig. 3, lines 11, 13)
13       return
     }}
     // deadlock avoidance
14   if (caller < p2t) {
       // a.o.:  caller < msr < p2t < p2t.msr
15     synchronized (caller) {
16       if (caller.cs == "TalkingWithMS") {
17         synchronized (msr) {
18           synchronized (p2t) {
19             if (p2t.cs != "TalkingWithMS") {
20               // T6' (Fig. 3, lines 11, 13)
                 }
21               else {
22                 synchronized (p2t.msr) {
23                   // T7' (Fig. 3, lines 14-16,
                     //                   18-24)
     }}}}}}}
24   else {
       // a.o.:  p2t < p2t.msr < caller < msr
25     // ...
   }}
```

Figure 5: Portion of an optimized handler

Figure 5 illustrates the structure of the optimized handlers for `OK` and `MESSAGE` messages. Integrating the synchronization logic into `doSuccessResponse` is straightforward. No locks are needed when determining the role that the message sender plays (lines 1 and 4). If it plays the caller role, the handler needs to lock only the message's SIP session (line 2). Otherwise, the handler may need both of the service execution's SIP sessions. The prescribed acquisition order (`a.o.`) requires the handler to first acquire the SIP session for the caller (line 5) and then, if needed, the SIP session for the media service (line 7).

In contrast, integrating the synchronization logic into the `doMessage` method makes the method significantly more complex. The handler must implement logic for two possible acquisition orders. We show the logic for just one of them (lines 14-23). The elided branch (lines 24, 25) is similar in complexity to the one shown, but is not a mirror image because the acquisition order affects the order of the nested conditionals.

Contract Version. For the contract version of the dating service, we wrote contracts declaring guard conditions and handles for the 11 transitions in Figure 1. The contracts are written in XML. They all follow the pattern of those illustrated in Table 1.

Discussion. Targeting the SNeF4SS API reduced both the amount and the complexity of the code that an application programmer needs to write for this example. The lines of code (LOC) for the message handlers in the optimized version (the shorter of the two manual versions) total to approximately 350. The message handler in the contract version contains approximately 200 LOC. Counting each contract as 5 LOC brings an estimate of LOC for the contract version to about 250.

The additional code in the manual versions accounts for most of the complexity of the code that the application programmer must write. As an indication of the complexity of the synchronization code in the optimized version, we tabulated the number of synchronized statements and their synchronized-nesting depths. Six of the message handlers in this version contain synchronized statements. Altogether, these 6 handlers contain 19 synchronized statements: eight at depth 1, five at depth 2, four at depth 3, and two at depth 4. The synchronization code dwarfs the business code in `doMessage` and `doBye`, which contain the depth-3 and depth-4 synchronized statements.

Synchronization code, whether encapsulated in a framework or written by the programmer, must infer the sessions that need to be acquired. Thus, the programmer must identify both guard conditions and handles, whether using the container API or the SNeF4SS API. When using SNeF4SS, the programmer expresses this information declaratively in a contract and leaves the framework to determine when and in what order to lock sessions. When using the container API, the programmer writes low-level operational code that infers the sessions needed and then acquires them in a consistent order. This code is significantly more complex than the code implementing the business logic. If mixed into the handlers, it obscures the business code and, as illustrated by Figure 5, makes tracing the business design to the code very difficult.

5. PERFORMANCE EVALUATION

The previous section presents the three implementations developed for the feasibility study and demonstrates that contract-based synchronization can provide benefits for design and development. This section describes the empirical evaluation we performed to assess whether dynamically interpreting contracts and using a generic negotiation protocol to acquire the needed sessions is practical in this domain.

We ran each of the three implementations of the dating service in the same test environment and under essentially the same loads and usage profiles. For each run, we calculated two widely-used performance metrics for SIP containers—throughput and response time.[9] We then compared how these metrics scale as the call rate increases. The remainder of this section provides details of the

[9]We also measured garbage collection rates and thread context switches, but do not report these results for lack of space.

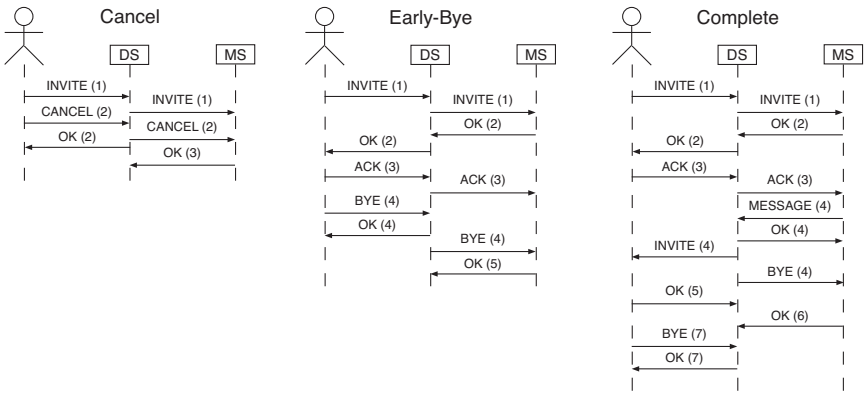


Figure 6: Sample message flows produced by three representative usage profiles

test environment (Sec. 5.1) and the test runs (Sec. 5.2), and then presents and discusses the performance results (Sec. 5.3 and 5.4, resp.).

5.1 Test environment

Our test environment contains five computers on a 100M switched ethernet network: two servers, one running the dating service and the other running the media service; two load generators, each simulating sets of callers; and one test coordinator, which reboots the servers prior to each run, triggers the load generators, and collects performance data.

Each server has a 64-bit Intel Xeon Quad Core 1.6GHz processor and 4G of memory. On each, we run 64-bit Linux (2.6 kernel), 64-bit JVM (Sun JDK 1.6.0 Update 14), and Sailfin SIP Servlet Container (1.0).[10] To reduce the impact of garbage collection, we configure the JVM using the guidelines in [19].

Each load generator has a 32-bit Intel Pentium IV 2.4GHz CPU and 1G of memory. The load generators run 32-bit Linux (2.6 kernel) and SIPp (3.1),[11] a widely used SIP load generating tool, to simulate concurrent callers. The hardware and software for the test coordinator are not material to the performance evaluation.

This hardware and software determines the *capacity* of the test environment. In test runs of the optimized version of the dating service, we observed that at about 100 concurrent service executions, the container starts to thrash. A call rate is unsustainable when messages from uncompleted service executions start to accumulate. We therefore regard 100 concurrent service executions as representing an overload for our configuration of the dating service. In contrast, our tests indicate that the optimized version can sustain 80 concurrent service executions indefinitely. We therefore regard 80 concurrent service executions as the expected load for our test configuration.

5.2 Test runs

A test run simulates a set of callers over a period of 300 seconds. Because performance is affected both by a caller's usage profile and by the call rate, a test run controls for both. We distinguish three usage profiles, which produce different message flows (Figure 6): *Cancel* (left) simulates a caller who hangs up immediately after placing the call; *Early Bye* (middle) simulates a caller who hangs up after her phone acknowledges the first `OK` response and before being paired with another caller; and *Complete Call* (right) simulates a caller who talks with another caller for two seconds and then hangs up. To indicate messages that are processed and sent by the same thread, we show thread number in parentheses. In practice, we expect callers to exhibit a mixture of these usage profiles, with the majority executing Complete Call. For a test run, therefore, we fixed the probabilities of the usage profiles at 5% Cancel, 5% Early Bye, and 90% Complete Call, randomly assigning a usage profile to each simulated caller with these probabilities.

To measure performance at expected load, we started each test run at 10 calls-per-second (cps) and increased the call rate by 5 cps every 10 seconds. Upon reaching expected load, or 80 concurrent service executions, we adjusted the call rate to keep the load just under 80. To measure differences in scalability, we increased the call rate by 5 cps every 10 seconds until 100 concurrent service executions was reached, instead of 80. We then adjusted the call rate to keep the number of concurrent service executions near 100. Each test starts with a warmup phase, during which containers are booted and data structures are initialized. No data is collected during this phase, as it is not material to performance of the service. We ran each test 10 times, each time for 300 seconds. The numbers reported are thus averages of 10 runs over these 300 seconds.

5.3 Performance results

The first manual version and the contract version of the dating service performed essentially the same in our test runs. For this reason and for lack of space, we present results of just the optimized and contract versions. Figure 7 plots results of measuring throughput (top two graphs) and response times[12] (bottom graph) for the optimized version (black solid) and the contract version (red dashed). We report throughput while operating under expected load (top) and under an overload (middle).

For throughput, we analyzed the system logs generated by the container, reporting the average number of messages processed per second during a run. Both implementations exhibited essentially the same throughput in the first 120 seconds. Prior to the 90th second, the throughput increased in a "step-like" fashion because the call rate increased by 5 every 10 seconds. Throughput leveled off as the number of concurrent service executions neared their targets: 80, which occurred at about the 70th second, under expected load; and 100, which occurred at about the 90th second, under the overload. The callers then started to maintain the target loads. In the case of expected load, throughput is essentially identical for the full 300 seconds. In the case of overload, both implementa-

[10] http://sailfin.dev.java.net

[11] http://sipp.sourceforge.net

[12] also referred to as *call-setup delay*

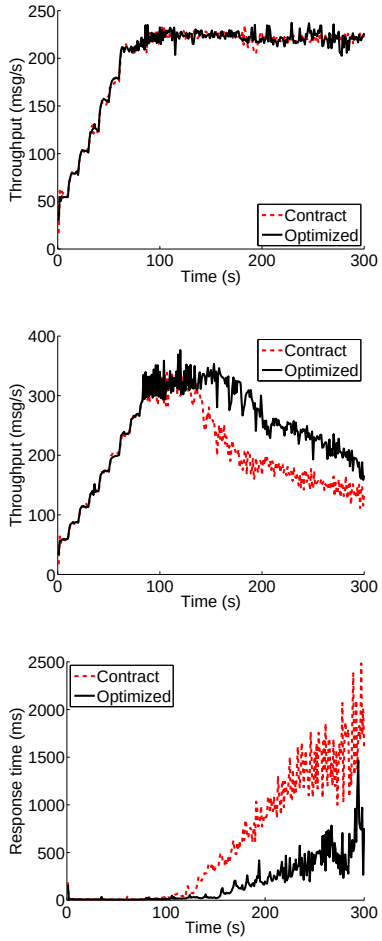


Figure 7: Performance comparison between optimized and contract versions (top: throughput at expected load; middle: throughput when overloaded; bottom: response time when overloaded

tions continue to exhibit similar throughput for approximately another 30 seconds. After this period, the throughput of the contract implementation begins to drop when overloaded, whereas that of the manual implementation remains relatively stable until the 160th second, at which time it also begins to drop.

For response time, we averaged all per-caller response times reported by SIPp, where a per-caller response time is the time elapsed between when the caller sends the `INVITE` message invoking a dating-service execution and receives an `OK` response from the dating service. As in the case of message throughput, we observed no noticeable difference in the performance of the optimized and contract versions as long as the system was not overloaded. But after about 30 seconds at near 100 concurrent service executions, we began to observe better measures for the optimized version. For the contract version, the response time is under 20 ms for 93% of the calls, and under 50 ms for 97%. For the optimized version, these percentages are 98% and 99%, respectively. Although the optimized version established more calls (5%) within 20 ms, both versions established most calls (97% and 99%) within 50 ms.

5.4 Discussion: Performance

Because the first manual version and the contract version use the same generic locking protocol, we expected them to exhibit the same scaling behavior. We attribute the finding that the actual measures observed for the contract version were as good as those for the first manual version to the fact that the first manual version needs to spend extra time in evaluating the guards in the `acquire()` method. Essentially, the overhead to evaluate the contract is similar to that to evaluate the guards.

The optimized version and the contract version exhibit the same scaling behavior when running near expected load. However, they exhibit different scaling behavior from the 120th second to the 160th second, when the container is overloaded. We believe this difference stems from two main sources. First, the contract version employs explicit locks to synchronize threads, whereas the manual version uses the Java synchronized statement. Locks provide the flexibility needed to separate the synchronization code from the business code, but incur some extra costs. Specifically, SNeF4SS creates a dedicated lock for each SIP session and maintains an internal mapping between a SIP session and its lock. Second, the implementation of resource numbering that we used for the contract version is fully generic in inferring the sessions needed and the locking order from the contract and a total order defined by the JVM; whereas the optimized version leverages an application-specific order relation and knowledge that certain attributes are assigned only once.

Of these two sources of differences, we believe that the latter is more significant and, moreover, that performance of a contract-based version can get even closer to that of the optimized version by adding information about single-assignment attributes to contracts. When processing a message from the caller, a negotiator first locks the caller's session; if the negotiator then finds that it also needs to lock the session for the media service and if the media service's session precedes the caller's session in hash-code order, it rolls back, unlocking the caller's session, then rolls forward, locking first the media service's session and then the caller's session, and finally rechecks the the guard condition in case it became stale during the back-off. In a dating-service execution, we expect approximately half of the messages bound for the service execution to be from the caller and about half from the media service. Moreover, the handler needs both the caller's session and the media service's session to process most messages. Thus, a negotiator frequently back-offs, even if no other negotiator is contending for a session it needs. Each back-off incurs extra overhead. The optimized version avoids this problem by checking single-assignment attributes outside of any critical regions. Thus, in future work, we intend to add tags to our contract language for SNeF4SS that declare single-assignment attributes and add another generic negotiator implementation to our current toolkit. The new negotiator will leverage single-assignment declarations to avoid acquiring locks unnecessarily.

6. CONCLUDING REMARKS

In summary, our case study confirms several software engineering benefits of using a contract-based synchronization framework and suggests that this approach to synchronization can be practical in the IPT domain. Performance of contract-based automatic synchronization was very close to that of an optimized manually-synchronized implementation when the system is resourced with hardware and software appropriate to the expected load.

We believe our success in this domain owes to several characteristics of IPT services. First, all of the sessions needed to process a message are reachable from the message-processing context, and navigation expressions provide a concise and compositional means for specifying how to dynamically infer these sets of sessions. Second, the short—under 0.2 ms [4]—bursty nature of message processing is conducive to synchronizing by acquiring all of

the needed sessions before performing any actions on them and releasing them all when all actions have been performed. Third, the information that a programmer needs to know to craft a synchronization contract is mostly available in the state machine that describes the business logic.

We are aware of several threats to validity of this study. First, the dating service is but one data point. While we designed it to be realistic and representative of the kinds of IPT services deployed today, we may have failed to capture some critical aspect of their complexity. Also, we focused our performance analysis on specific usage profiles and in specific proportions. An earlier analysis, however, using just Call Complete, the most complex profile, produced similar results. Second, we chose to use a locking protocol based on a specific policy, resource numbering. The comparison might differ if the manual implementation used a more *ad hoc* policy. That said, it would have been difficult to fairly compare the contract-based synchronization to manual synchronization if each used a fundamentally different locking protocol. Finally, it is possible that, despite our best efforts, some bias toward our approach entered into the implementations of the different versions of the dating service developed for the study. We address this threat by making the source code public, thereby allowing other researchers to replicate our results.

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.

7. REFERENCES

[1] R. Behrends and R. E. K. Stirewalt. The Universe Model: an approach for improving the modularity and reliability of concurrent programs. In *Proc. FSE'00: ACM SIGSOFT 8th Intl. Symp. Foundations of Softw. Eng.*, pages 20–29, San Diego, CA, USA, November 2000. ACM.

[2] A. J. Bernstein and P. M. Lewis. *Concurrency in programming and database systems*. Jones and Bartlett Publishers, Inc., USA, 1993.

[3] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. ASE '04: 19th IEEE Intl. Conf. Automated Softw. Eng.*, pages 248–257, Lawrence, KA, USA, September 2004. IEEE Computer Society.

[4] M. Cortes, J. Ensor, and J. Esteban. On SIP performance. *Bell Labs Technical J.*, 9(3):155–172, Jan 2004.

[5] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications*, 28(11):65–69, 1990.

[6] C. M. Fleiner and M. Philippsen. Fair multi-branch locking of several locks. In *Proc. ICPDCS'97: Intl. Conf. Parallel and Distributed Computing Systems*, pages 537–545, Washington, DC, USA, October 1997.

[7] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] V. Gurbani, L. Jagadeesan, and V. Mendiratta. Characterizing session initiation protocol (SIP) network performance and reliability. In M. Malek, E. Nett, and N. Suri, editors, *Service Availability*, volume 3694 of *Lecture Notes in Computer Science*, pages 196–211. Springer Berlin / Heidelberg, 2005.

[9] A. Hamie, J. Howse, and S. Kent. Navigation expressions in object-oriented modelling. In E. Astesiano, editor, *Proc. FASE'98: Fundamental Approaches to Softw. Eng.*, volume 1382 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin / Heidelberg, 1998.

[10] P. B. Hansen. *Operating System Principles*. Prentice Hall, USA, 1973.

[11] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems J.*, 7(2):74–84, 1968.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] Y. Huang, E. Cheung, L. K. Dillon, and R. E. K. Stirewalt. A thread synchronization model for SIP servlet containers. In *Proc. IPTComm '09: 3rd Intl. Conf. Principles, Systems and Applications of IP Telecommunications*, pages 1–12, Atlanta, GA, USA, 2009. ACM.

[14] Y. Huang, L. K. Dillon, and R. E. Stirewalt. On mechanisms for deadlock avoidance in SIP servlet containers. In *Proc. IPTComm '08: 2nd Intl. Conf. Principles, Systems and Applications of IP Telecommunications*, pages 196–216, Heidelberg, September 2008. Springer-Verlag.

[15] Y. Huang, L. K. Dillon, and R. E. K. Stirewalt. Prototyping synchronization policies for existing programs. In *Proc. ICPC'09: Intl. Conf. Program Comprehension*, pages 289–290. IEEE Computer Society, 2009.

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1988.

[17] E. M. Nahum, J. Tracey, and C. P. Wright. Evaluating SIP server performance. In *Proc. SIGMETRICS '07: ACM Intl. Conf. Measurement and Modeling of Computer Systems*, pages 349–350, San Diego, CA, USA, June 2007. ACM.

[18] T. M. Smith and G. W. Bond. ECharts for SIP servlets: a state-machine programming environment for voip applications. In *Proc. IPTComm '07: 1st Intl. Conf. Principles, Systems and Applications of IP Telecommunications*, pages 89–98, New York, NY, USA, July 2007. ACM.

[19] B. Van Den Bossche, F. De Turck, B. Dhoedt, P. Demeester, G. Maas, J. Moreels, B. Van Vlerken, and T. Pollet. J2EE-based middleware for low latency service enabling platforms. In *Proc. GLOBECOM '06: Global Telecommunications Conf.*, pages 1–6, San Francisco, CA, USA, November 2006.

[20] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In *Proc. OOPSLA '96: 11th ACM SIGPLAN Conf. Object-oriented programming, systems, languages, and applications*, pages 359–369, San Jose, CA, USA, October 1996. ACM.

[21] S. Wanke, M. Scharf, S. Kiesel, and S. Wahl. Measurement of the SIP parsing performance in the SIP express router. In *Proc. EUNICE'07: 13th EUNICE Open European Summer School (IFIP TC6.6 Conf. Dependable and Adaptable Networks and Services)*, pages 103–110, Enschede, The Netherlands, 2007. Springer-Verlag.

[22] J. Wilkiewicz and M. Kulkarni. JSR 289: SIP servlet specification v1.1. http://jcp.org/aboutJava/communityprocess/final/jsr289.

[23] R. Wirfs-Brock and A. McKean. *Object Design: Roles, Responsibilities and Collaborations*. Addison–Wesley, 2003.

[24] P. Zave and E. Cheung. Compositional control of IP media. *IEEE Trans. Softw. Eng.*, 35(1):46–66, 2009.

[25] D. Zöbel. The deadlock problem: a classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17(4):6–15, 1983.