Stable Availability under Denial of Service Attacks through Formal Patterns*

Jonas Eckhardt[1,2,3], Tobias Mühlbauer[1,2,3], Musab AlTurki[4], José Meseguer[4], and Martin Wirsing[1,5]

[1] Ludwig Maximilian University of Munich
[2] Technical University of Munich
[3] University of Augsburg
[4] University of Illinois at Urbana-Champaign
[5] IMDEA Software

Abstract. Availability is an important security property for Internet services and a key ingredient of most service level agreements. It can be compromised by distributed Denial of Service (DoS) attacks. In this work we propose a formal pattern-based approach to study defense mechanisms against DoS attacks. We enhance pattern descriptions with formal models that allow the designer to give guarantees on the behavior of the proposed solution. The underlying executable specification formalism we use is the rewriting logic language Maude and its real-time and probabilistic extensions. We introduce the notion of stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. Then we present two formal patterns which can serve as defenses against DoS attacks: the Adaptive Selective Verification (ASV) pattern, which enhances a communication protocol with a defense mechanism, and the Server Replicator (SR) pattern, which provisions additional resources on demand. However, ASV achieves availability without stability, and SR cannot achieve stable availability at a reasonable cost. As a main result we show, by statistical model checking with the PVeStA tool, that the composition of both patterns yields a new improved pattern which guarantees stable availability at a reasonable cost.

Keywords: formal patterns, meta-object pattern, rewriting logic, availability, denial of service, statistical model checking, cloud computing.

1 Introduction

On December 8, 2010 at 07:53 AM EDT, MasterCard issued a statement that "MasterCard is experiencing heavy traffic on its external corporate website [...]. There is no impact whatsoever on our cardholders ability to use their cards for secure transactions" [19]. In fact, by that time, a distributed Denial of Service

* This work has been partially sponsored by the Software Engineering Elite Graduate Program, the EU-funded projects FP7-257414 ASCENS and FP7-256980 NESSoS, and AFOSR Grant FA8750-11-2-0084. The fourth author was also partially supported by the "Programa de Apoyo a la Investigación y Desarrollo" (PAID-02-11) of the Universitat Politècnica de València.

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 78–93, 2012.
© Springer-Verlag Berlin Heidelberg 2012

attack (DoS) brought the website down and made their web presence unavailable for most customers for several hours. Availability is an important security property for Internet services and a key ingredient of most service level agreements.

DoS defense mechanisms help maintaining availability; nevertheless even when equipped with defense mechanisms, systems will typically show performance degradation. Thus, one of the goals of security measures is to achieve stable availability, which means that with very high probability service quality remains very close to a constant quantity, which does not change over time, regardless of how bad the DoS attack can get. Cloud Computing, by offering the possibility of dynamic resource allocation, can be used to leverage stable availability when combined with DoS defense mechanisms. Service-oriented systems such as the MasterCard service are distributed systems operating in a dynamically changing environment. They need to cope with changing numbers of user demands and with hostile attacks. To be used/operated safely, services have to satisfy functional as well as non-functional requirements and it is not a priori clear what is the best realization of a service in each particular situation. Model-driven approaches to service development offer the possibility of tackling these issues at a high level of abstraction during early stages of system analysis and design. In particular, design patterns have been successfully used for improving programming solutions in several domains, including object-orientation [13], service-oriented computing [17,12] and security [25]. Patterns are general, reusable solutions to commonly occurring problems in software design; they clearly define the programming context, the problem and the advantages and disadvantages of design solutions (see e.g., [13,25]).

In this work, we introduce formal patterns which, in addition to "normal" patterns, come with formal guarantees and enable automated pattern composition, often resulting in semi-automatic construction of new models with improved properties. We use this pattern-based approach to study defense mechanism against DoS attacks in a model-based setting. We present two formal patterns which can serve as defenses against DoS attacks: the Adaptive Selective Verification (ASV) [15] pattern defending against DoS attacks, and the Server Replicator (SR) pattern in a cloud setting. As underlying executable specification formalism we use the rewriting logic language Maude and its real-time and probabilistic extensions. The ASV protocol is a well-known defense against DoS attacks in the typical situation that clients and attackers use a shared channel where neither the attacker nor the client have full control over the communication channel [15]. The ASV protocol adapts to increasingly severe DoS attacks and provides improved availability. However, it cannot provide stable availability. By replicating servers one can dynamically provision more resources to adapt to high demand situations and achieve stable availability; but the cost of provisioned servers drastically increases in a DoS attack situation. These two patterns are modeled in Maude and then formally composed to obtain the new improved $ASV^{+}SR$ pattern. As a main result we show, by analyzing the quantitative properties of $ASV^{+}SR$ with the statistical model checker PVeStA, that $ASV^{+}SR$ guarantees stable availability at a reasonable cost.

Outline. The paper is structured as follows: Sect. 2 introduces the notion of stable availability and gives a short account of the prerequisites on rewriting logic, Maude, and the statistical model checking of quantitative properties with the PVeSTa tool in Maude. In Sect. 3 we present the concept of formal patterns and give three examples: (i) the general meta-object pattern (Sect. 3.1), (ii) the ASV pattern (Sect. 3.2), and (iii) the SR pattern (Sect. 3.3). In Sect. 4 we present the ASV^{+}SR composition pattern and validate the properties of the composed system using the PVeSTa tool. We conclude by discussing related work, summarizing our results and sketching further work.

2 Prerequisites

2.1 Rewriting Logic and Maude

Rewriting logic [21] is a simple computational logic to specify concurrent and object-oriented systems as *rewrite theories*, that is, as triples (Σ, E, R), where (Σ, E) is an *order-sorted equational theory* with syntax and type structure specified by the signature Σ, and with (possibly conditional) Σ-equations E; and where R is a set of (possibly conditional) *rewrite rules* of the form $t \rightarrow t'$ *if cond*, with t, t' Σ-terms, and *cond* the rule's condition.

The Maude system [9] executes rewrite theories, with a self-explanatory typewriter syntax almost isomorphic to the mathematical syntax. The key concept in Maude is that of a module. An object-oriented module defines a class named K and attributes $a_1 \ldots, a_n$. An object o in a given state can be represented as a term of the form $\langle o : K \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ where $v_1 \ldots, v_n$ are the corresponding values stored in those attributes. A *message* addressed to object o with contents d can be represented as a term $(o \leftarrow d)$; and all messages in a system are then terms of sort *Message*. The distributed systems we consider in this paper are systems, made up of objects that communicate with each other by asynchronous message passing. The *distributed state* of such a system is a *multiset* or "soup" of objects and messages, called a *configuration*. Mathematically, this is specified by declaring a sort *Configuration* with subsort inclusions $\mathit{Object}, \mathit{Message} < \mathit{Configuration}$, and an associative and commutative multiset union operator with empty syntax: $__ : \mathit{Configuration}\ \mathit{Configuration} \longrightarrow \mathit{Configuration}$ and with identity element *null*.

For example, a simple client class may have name *Client*; a simple server class may have name *Server* and an attribute *bf* for storing the received messages in a buffer. In a simple request-response message exchange pattern (cf. [27]) a client c sends request packets ($\mathit{req}(\mathrm{c})$) to the server. In response, the server sends response packets (ack) back to the client. The following term defines a configuration containing one server object s with a request from *c1* in the buffer, two client objects and one message addressed to *c1*.

$$\langle s : \mathit{Server} \mid \mathit{bf} : \mathit{req}(c1) \rangle\ \langle c1 : \mathit{Client} \mid \rangle\ \langle c2 : \mathit{Client} \mid \rangle\ (c1 \leftarrow \mathit{ack})$$

The following rewrite rule defines the reaction of any server object s upon receipt of a request $(s \leftarrow \mathit{req}(c))$ from any client c.

$$\textbf{rl}\ (s \leftarrow req(c))\ \langle s : Server \mid bf : b \rangle\ \rightarrow \langle s : Server \mid bf : b\ req(c) \rangle\ (c \leftarrow ack)\ .$$

The server adds $req(c)$ to the buffer and sends an acknowledgement $(c \leftarrow ack)$ back to the client c. Although not illustrated by the rule above, upon receiving message an object can send several messages to other objects, and can create new objects.

Rewriting logic can naturally model concurrent systems, which can be both *real-time* and *probabilistic*. Real-Time systems are supported by rewrite theories (Σ, E, R) whose underlying equational theory (Σ, E) includes among its types an algebraic data type *Time* representing time instants (which may be either discrete or continuous), and whose global states are pairs of the form (t, r), with t a term representing a "discrete" state, and r a time value of sort *Time* representing the global clock. The rewrite rules in R can then be either *instantaneous* rules, which do not change the global clock, or *tick* rules, which advance the global time (see [24]). Probabilistic concurrent systems, which may also be real-time systems, are modeled by *probabilistic rewrite rules* of the form

$$l : t(\boldsymbol{x}) \rightarrow t'(\boldsymbol{x}, \boldsymbol{y})\ \ \textit{if}\ \ cond(\boldsymbol{x})\ \ \textit{with probability}\ \ \boldsymbol{y} := \pi_l(\boldsymbol{x})$$

where the righthand side term t' has new variables $\boldsymbol{y}$ disjoint from the variables $\boldsymbol{x}$ appearing in t which make the application of the rule non-deterministic. The probabilistic nature of the rule is expressed by the probability distribution $\pi_l(\boldsymbol{x})$ with which values for the extra variables $\boldsymbol{y}$ are chosen; where $\pi_l(\boldsymbol{x})$ is in general not fixed, but parametric on the righthand side variables $\boldsymbol{x}$. In this paper, we use the PMaude [4] notation for probabilistic rewrite rules.

A *parameterized module* $M[X :: P]$ has a formal parameter X satisfying a parameter theory P; M can be instantiated by another module Q via a theory interpretation $V : P \longrightarrow Q$, called a *view*, with the usual pushout semantics (see [9]). We denote the resulting module by $M[V]$ or shorter by $M[Q]$ if V is clear from the context.

2.2 Statistical Model Checking of Quantitative Properties

Temporal logic properties of a probabilistic system can be model checked either by exact model checking algorithms or, in an approximate but more scalable way, by *statistical model checking* (see, e.g., [26,29,4]). The idea of statistical model checking is to verify the satisfaction of a temporal logic property by statistical methods up to a user-specified level of statistical confidence. For this, a large enough number of Monte-Carlo simulations of the system are performed, and the formula is evaluated on each of the simulations.

Current statistical model checking algorithms assume that the system is purely probabilistic, i.e., that there is no nondeterminism in the choice of transitions. Using the methodology presented in [4] and further extended in this work to the case of reflective "Russian dolls" architectures, a wide class of object-oriented probabilistic real-time distributed systems can be expressed as purely probabilistic systems. In particular, all the distributed systems considered in this paper fall within this broad class.

To analyze the behavior of systems with respect to quantitative properties related to performance and QoS, a *quantitative* temporal logic, where the result of evaluating a formula is not a Boolean true/false value, but a real number, can be used. For this purposes we use the QuaTEx quantitative temporal logic [4], and the PVeStA [6] parallelization of its associated VeStA tool and model checking algorithm [4]. In Sect. 4.2 we will present several QuaTEx expressions formalizing crucial quantitative properties related to DoS protection and will model check them in PVeStA. We refer the reader to [4] for a detailed description of QuaTEx expressions and their model checking algorithm. In this paper, we will compute the expected value of a path expression based on definitions of the form $F(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{EXP} \textbf{ else } \bigcirc (F(t))$, where $\bigcirc$ is the next operator, $\mathit{time}()$ is a state function returning the global time, and EXP is a real-valued state function.

2.3 Stable Availability

Availability is a key security property by which a system remains available to its users under some conditions. This property can be compromised by a DoS attack, which may render a system unavailable in practice. What all DoS defense mechanisms have in common is the goal of protecting a system's availability properties in the face of a DoS attack. But availability properties are *quantitative* properties: some DoS defense mechanisms may provide better QoS properties and therefore better availability properties than others. In fact, even when protected against DoS, performance degradation will typically be experienced in some aspects of system behavior such as, for example, the average Time To Service (TTS) experienced by clients, the success ratio with which clients manage to communicate with their server, or the average bandwidth (or some other cost measure) that a client needs to spend to successfully communicate with its server. Obviously, an ideal DoS protection scheme is one that renders the system to a large extent *impervious* to the DoS attack, no matter how bad the attack can get.[1] That is, up to some acceptable and constant performance degradation, the system behaves in a "business as usual" manner: as if no attack had taken place, even when in fact the attack worsens over time. We call this property *stable availability.* As we shall show in Sect. 4, stable availability can be achieved in some cases by using an appropriate meta-object architecture for DoS protection.

More precisely, the stable availability of a system assumes a shared channel [14], where DoS attackers can at most monopolize a maximum percentage of the overall bandwidth. Under these circumstances, stable availability is formulated as a requirement parameterized by explicitly specified and *quantifiable* availability properties such as, for example, TTS, success ratio, average bandwidth, and so on. The system is then said to be *stably available* with respect to the specified

[1] In the shared channel model of [14], attackers can have a potentially very large but not absolute share of the overall bandwidth, so that honest users will still have some bandwidth available. This is a realistic assumption in most situations, and a key difference between DoS attackers and Dolev-Yao attackers, who, having full control of the channel, can always destroy *all* honest user messages.

quantities if and only if, with very high probability, each such quantity q remains very close (up to fixed bounds ε) to a *threshold* θ ($| q - \varepsilon | < \theta$), which does not change over time, regardless of how bad the DoS attack can get within the bounds allowed by the shared channel assumption.

3 Formal Patterns

Pattern-based approaches have been successfully introduced to help developers choose appropriate design and programming solutions [13]. However, these informal patterns typically offer limited help for assessing the required functional and non-functional properties. This is particularly important in the case of distributed systems, which are notoriously hard to build, test, and verify. To ameliorate this problem we are proposing to enhance pattern descriptions with executable specifications that can support the mathematical analysis of qualitative and quantitative properties; thus allowing the designer to give guarantees on the behavior of the proposed solution.

A formal pattern *Pat* is structured in the usual way (cf. e.g. [25,12]) in context, problem, solution, advantages and shortcomings (and other features such as forces, related patterns which we mostly omit here for simplicity); but instead of using UML or Java we describe the solution formally as a parameterized module $M[S]$ in Maude (with parameter theory S) and draw many of the advantages and shortcomings of a pattern from formal analyses. Moreover, the context typically describes also the assumptions of the parameter theory S.

Pattern composition $Pat + Pat'$ of two patterns Pat and Pat' formalized as parameterized Maude modules $P[S]$ and $P'[S']$ can be achieved by an appropriate “parameterized view” (see [9]) connecting both patterns. For example, we may instantiate S' to $P[S]$, yielding the composed pattern $P'[P[S]]$. The problem statement and context of *Pat+ Pat'* can then be systematically derived from those of *Pat* and *Pat′*.

In the following we present several formal patterns which can be very useful to make distributed systems adaptable to changing and potentially hostile environments, and show how to design and analyze such systems in a modular and predictable way.

3.1 The Meta-object Pattern

Concurrency is not the only challenge for distributed systems: *adaptation* is just as challenging, since many distributed systems need to function in highly unpredictable and potentially hostile environments such as the Internet, and need to satisfy safety, real-time and Quality of Service (QoS) requirements which are essential for their proper behavior. To meet these adaptation challenges and the associated requirements, a modular approach based on *meta-objects* can be extremely useful. A meta-object pattern *MO* is defined as follows:

Context. A concurrent and distributed object-based system.

Problem. How can the communication behavior of one or several objects be dynamically *mediated/adapted/controled* for some specific purposes?

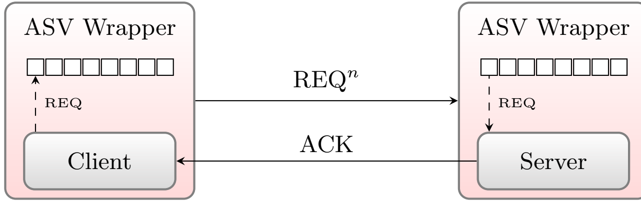


Fig. 1. Application of the ASV meta-object on a client-server request-response service

Solution. A meta-object is an object which dynamically *mediates/adapts/controls* the communication behavior of one or several objects under it. In rewriting logic, a meta-object can be specified as an object of the form $\langle o : K \mid conf : c, a_1 : v_1, \ldots, a_n : v_n \rangle$, where c is a term of sort *Configuration*, and all other $v_1 \ldots, v_n$ are not configuration terms. The configuration c contains the object or objects that the meta-object o controls. Thus the parameterized module $MO[X]$ introduces the meta-object constructor; the parameter X specifies the sorts $s_1, \ldots, s_n$ and attributes $a_1, \ldots, a_n$ of the controlled system.

Advantages and Shortcomings. MO defines a general control and wrapper architecture; but may add communication indirection and the requirement for language specific object visibility.

There are many different MO patterns: If c contains a single object, the meta-object o is sometimes called an *onion-skin* meta-object [2], because o itself could be wrapped inside another meta-object, and so on, like the skin layers in an onion. More generally, c may not only contain several objects $o_1 \ldots, o_m$ inside: it may also be the case that some of these o_i are themselves meta-objects that contain other objects, which may again be meta-objects, and so on. That is, the more general reflective meta-object architectures are so-called "Russian dolls" architectures [22], because each meta-object can be viewed as a Russian doll which contains other dolls inside, which again may contain other dolls, and so on.

In the following we will present meta-object patterns that illustrate both the onion-skin case, and the general Russian dolls case.

3.2 The ASV DoS Protection Meta-object Pattern

The ASV protocol [15] is a cost-based, DoS-resistant protocol where bandwidth is used as currency by a server to discriminate between good and malicious users; that is, honest clients spend more bandwidth by replicating their messages.

Context. Client-server request-reply system under DoS attack, shared channel attacker model [14].

Problem. How can the system be protected against DoS attacks?

Solution. Informally described, the server and the clients are wrapped by meta-objects with the following key features: The client wrappers attempt to adapt to the current level of attack by exponentially replicating the client requests up to a fixed bound. The server wrapper adapts to the level of the attack by dropping randomly packets, with a higher probability as the attack becomes more severe. Only the remaining requests are processed by the server.

Fig. 1 illustrates the ASV meta-object pattern.

A first modularized formalization of the ASV protocol was given by AlTurki in [5]. In this work we extend this specification by making its modularization more explicit using parametrized modules. The modularized ASV meta-object specification ($ASV[S]$) is parametric in the client-server system S. In particular, we assume that S indicates the maximal load *maxLoad* per server. Clients have a time-out window (which is set to the expected worst case round-trip delay between the client and the server) and a replication threshold, i.e. the maximum number of times a client tries to send requests to the server before it gives up.

We present only the behavior of the server wrapper in a little more detail. The wrapper counts the incoming requests and places them in a buffer *buf*. If the buffer length of the servers exceeds *maxLoad*, a coin is tossed to decide whether an incoming message should be dropped or not, i.e., it is randomly decided according to a Bernoulli distribution *Ber* with success probability $\mathit{floor}(\mathit{maxLoad})/(\mathit{cnt}+1.0)$. If the message is not dropped, a position of *buf* is randomly chosen with uniform distribution *Uni* and the new message is stored at this position (replacing another message).

$$
\begin{array}{l}
\textbf{crl } (s \leftarrow c)\ \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt}, \mathit{buf} : L \rangle \ \rightarrow \\
\ \textbf{if } (y_2) \textbf{ then } \ \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt} + 1.0, \mathit{buf} : L[y_1] := c \rangle \\
\ \textbf{else } \langle s : \mathit{asvServer} \mid \mathit{count} : \mathit{cnt} + 1.0, \mathit{buf} : L \rangle \textbf{ fi} \\
\textbf{if } \mathit{float}(L.\mathit{size}) \geq \mathit{floor}(\mathit{maxLoad}) \\
\textbf{with probability } y_1 := \mathit{Uni}(L.\mathit{size}) \\
\ \textbf{and } y_2 := \mathit{Ber}(\mathit{floor}\,(\mathit{maxLoad})\,/(\mathit{cnt} + 1.0)).
\end{array}
$$

In addition, the server wrapper periodically empties its buffer and sends the contents to the wrapped server. Answers of the server are forwarded to the client.

Advantages & Shortcomings. The ASV protocol has remarkably good properties, such as closely approximating *omniscience* [15]: although only local knowledge is used by each protocol participant, ASV's emergent behavior closely approximates the behavior of an idealized DoS defense protocol in which all relevant parameters describing the state of the attack are instantaneously known to all participants. However, it cannot provide stable availability [11,23].

3.3 The Server Replicator Meta-object

In high-demand situations, Cloud-based services can benefit from the scalability of the Cloud, i.e., from the dynamic allocation of resources. The **S**erver **R**eplicator meta-object (SR) is a simple pattern that adapts to high-demand situations by leveraging the scalability of the Cloud [11,23].

Context. Client-server request-reply system; possibility of provisioning additional resources.

Problem. How can the system adapt to an increasing amount of requests, e.g., caused by a DoS attack?

Solution. The SR wraps instances of servers that provide a service, dynamically provisions new such instances to adapt to an increasing load, and distributes incoming requests among them.

The meta-object SR ($SR[S]$) is parametric in the client-server system S, whose servers (of class (*Server*)) it creates instances of. In order to be replicable, the servers in S need to fulfill a theory which specifies how a server instance is created (*replicate*) and initialized (*init*); and how many requests it can handle within a specific timeframe (*maxLoadPerServer*). Additional parameters in S specify a replication strategy which determines the *overloading factor* which must be exceeded before a new server is provisioned.

SR performs the following tasks:

Provisioning New Instances of the Server. SR periodically evaluates its replication strategy and, if necessary, spawns a new server instance. The behavior of spawning a new server is described by the rewrite rule

$$\begin{aligned}&\textbf{crl}\ (sr \leftarrow spawnServer)\ \langle sr : ServerReplicator \mid server\text{-}list : SL, config : NG\ C\rangle \\ &\ \rightarrow \langle sr : ServerReplicator \mid server\text{-}list : (sa; SL), \\ &\qquad config : (NG.next)\ C\ replicate(sa)\ init(sa)\rangle \\ &\textbf{if}\ sa := NG.new\ .\end{aligned}$$

Removing Instances of the Server. SR winds down the number of replicated servers when the load decreases. We do not model this behavior. One solution would be to synchronize the communication between SR and a server instance by using a buffer. SR sets a server instance it wants to remove as inactive and no longer forwards requests to it. When an inactive server has processed all requests in its buffer, it removes itself from the configuration.

Distribution of Incoming Messages. SR randomly distributes incoming requests among its servers in a uniform way using the rule

$$\begin{aligned}&\textbf{rl}\ (sr \leftarrow CO)\ \langle sr : ServerReplicator \mid server\text{-}list : SL, config : C\rangle\ \rightarrow \\ &\langle sr : ServerReplicator \mid server\text{-}list : SL, config : (y_1 \leftarrow CO)\ C\rangle \\ &\textbf{with probability}\ y_1 := Random(SL)\ .\end{aligned}$$

where *Random* randomly chooses a server from a list of servers.

Forwarding Messages to the Outside. Additionally, SR specifies rules to forward messages that address client objects located outside its boundary.

Advantages & Shortcomings. SR can provide stable availability. However, the cost of provisioning servers drastically increases in high-demand situations.

4 Stable Availability under Denial of Service Attacks through Formal Patterns

How can meta-object patterns be used to make a Cloud-based client-server request-response service resilient to DoS attacks with minimum performance degradation, that is, achieving in fact stable availability at reasonable cost?

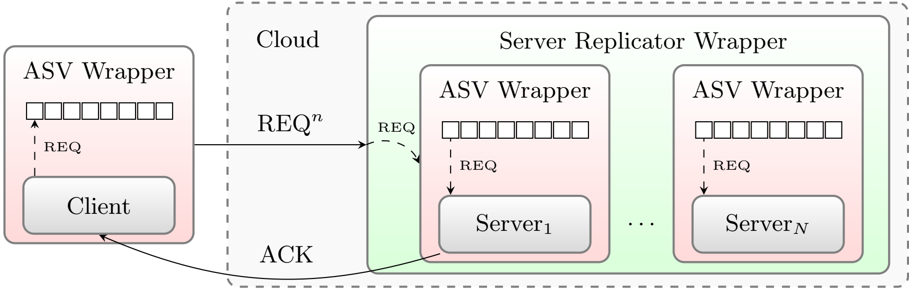


Fig. 2. Application of the ASV$^+$SR meta-object composition on a Cloud-based client-server request-response service

We propose to investigate this question by composing a client-server system S with appropriate meta-object patterns.

4.1 ASV$^+$SR Meta-object Composition Pattern

Combining the ASV and SR meta-object patterns into ASV$^+$SR enables us to overcome their respective shortcomings while keeping their advantages.

Context. Client-server request-reply system under DoS attack, shared channel attacker model [14]; possibility of provisioning additional resources.

Problem. How can the system be protected against the DoS attack and provide stable availability at reasonable cost?

Solution. The application of the meta-object composition on S, $SR[ASV[S], \rho]$, (where ρ maps the formal parameter ($Server$) to ($asvServer$) and ($maxloadServer$) to ($maxLoad$)) protects the service against DoS attacks in two dimensions of adaptation: (i) the ASV mechanism; and (ii) the SR replication mechanism. Fig. 2 gives an overview of the composition.

We define the factor k that proportionally adjusts the degree of ASV protection in the meta-object composition, i.e., k reflects how much the ASV mechanism is used compared to the SR replication mechanism. An overloading factor of $k = 1$ means that the ASV mechanism remains nearly unused, while an overloading factor of $k = \infty$ means that the replication mechanism is unused. Thus, we propose an overloading factor of $1 < k < \infty$.

The replication strategy for computing the number of server replicas γ is defined as

$$\gamma(m, t) = \max\left(1, \frac{m}{maxLoadPerServer(t) \cdot k}\right)$$

where m denotes the number of messages that have been received by the SR up to time t; and $maxLoadPerServer(t)$ is defined as

$$maxLoadPerServer(t) = \left\lfloor \frac{t}{T} \right\rfloor \cdot maxLoad_S$$

where T is the ASV server timeout period and $maxLoad_S$ denotes the buffer size of the ASV server.

Advantages & Shortcomings. We will show that the ASV$^+$SR composition provides stable availability under DoS attacks at the cost of provisioning a predictable amount of instantiated servers given by the overload factor.

4.2 Statistical Model Checking Analysis

We use the Maude-based specification of the ASV$^+$SR meta-object pattern with a client-server system to perform parallelized statistical quantitative model checking on 20 to 40 cluster nodes using PVeStA. The expected values of the following QuaTEx path expressions were computed with a 99% confidence interval of size at most 0.01:

Client Success Ratio. The client success ratio defines the ratio of clients that receive an acknowledgement from the server.

$$\begin{aligned} \mathit{successRatio}(t) = \ & \mathbf{if}\ \mathit{time}() > t\ \mathbf{then}\ \mathit{countSuccessful}()/\mathit{countClients}() \\ & \mathbf{else}\ \bigcirc\,(\mathit{successRatio}(t)) \end{aligned}$$

where *countClients*() and *countSuccessful*() respectively count the total number of clients, and the number of clients with *"connected"* status.

Average TTS. The average TTS is the average time it takes for a successful client to receive an acknowledgement from the server.

$$\begin{aligned} \mathit{avgTTS}(t) = \ & \mathbf{if}\ \mathit{time}() > t\ \mathbf{then}\ \mathit{sumTTS}()/\mathit{countSuccessful}() \\ & \mathbf{else}\ \bigcirc\,(\mathit{avgTTS}(t)) \end{aligned}$$

where *sumTTS*() is the sum of the TTS values of all successful clients.

Number of Servers. The number of servers represents the number of ASV servers that are spawned by the SR meta-object.

$$\begin{aligned} \mathit{servers}(t) = \ & \mathbf{if}\ \mathit{time}() > t\ \mathbf{then}\ \mathit{countServers}() \\ & \mathbf{else}\ \bigcirc\,(\mathit{servers}(t)) \end{aligned}$$

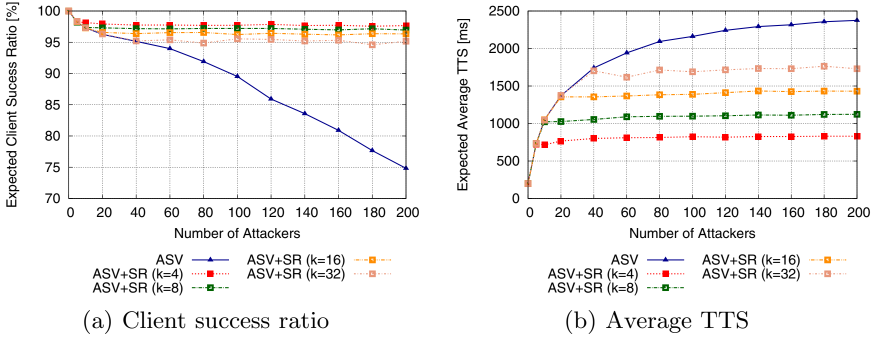
where *countServers*() is the number of replicated servers.

For statistical model checking purposes we set the parameters of the ASV and SR meta-objects as follows:

ASV. The mean server processing rate is set to 600 packets per second, the timeout window of the clients to 0.4 seconds, the retrial span of the clients to 7, and the client arrival rate to 0.08.

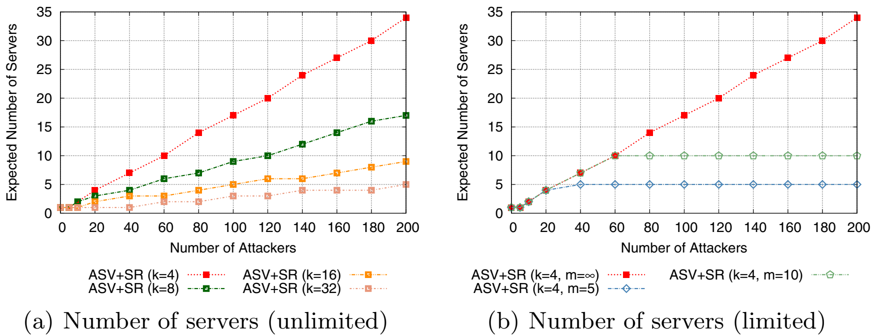
SR. The check period is set to 0.01 seconds and we vary the overloading factor k (4, 8, 16, 32). Forward and replication delays are not considered in our experiments.

The properties are checked for a varying number of attackers (1 to 200). Each attacker issues 400 fake requests per second. It is of note that 1.5 attackers already overwhelm a single server. The values of the ASV and attack parameters correspond to the values chosen in [7,15]. Additionally, an initial generation delay of 0.05 seconds is introduced and the duration of a simulation is set to 30 seconds.



(a) Client success ratio (b) Average TTS

Fig. 3. Performance of the ASV$^+$SR protocol with a varying load factor k and no resource bounds

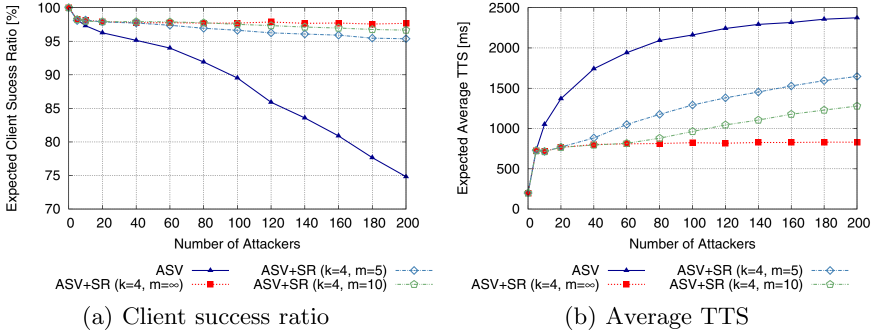


(a) Number of servers (unlimited) (b) Number of servers (limited)

Fig. 4. Expected number of servers using the ASV$^+$SR protocol

In the following, we will consider two general cases in which the SR can provision: (i) an unlimited number of servers, and (ii) servers up to a limit m of 5 or 10 servers, because, due to economical and physical restrictions, resources are limited. The results in (i) will indicate how many servers are needed to provide stable service guarantees, while the results in (ii) will indicate what service guarantees can still be given with limited resources.

Unlimited Resources. Fig. 3 shows the model checking results for a varying overloading factor k with no resource limits. As indicated by Fig. 3(a), ASV$^+$SR can sustain the expected client success ratio at a certain percentage. Even for an overloading factor of $k = 32$, a success ratio around 95% can be achieved. Compared to an overloading factor of $k = 4$, a 7-fold decrease in provisioned servers is observed (Fig. 4(a)), achieving a stable success ratio of only around 3% less. Fig. 3(b) shows that the same is true for the average TTS. ASV$^+$SR outperforms the ASV protocol, and furthermore achieves stable availability, for all performance indicators. However, this comes at the cost of provisioning new servers. Fig. 4(a) shows how many servers are provisioned. The results indicate that the factor k defines a trade-off between the cost and the performance of stable availability. SR by itself ($k = 1$) with unlimited resources (not shown in the figures) would



(a) Client success ratio (b) Average TTS

Fig. 5. Performance of the ASV$^+$SR protocol with a load factor of $k = 4$ and limited resources

provide stable availability at a level as if no attack has happened, but would provision 134 servers for 200 attackers. Note that fluctuations in the results, e.g., the average TTS in case of 60 attackers being lower than the average TTS in case of 40 attackers, are due to the provisioning of a discrete number of servers.

Limited Resources. Fig. 5 shows the model checking results for an overloading factor of $k = 4$ and a limit m of either 5 or 10 servers that the SR meta-object can provision. As indicated by Fig. 5(a), the success ratio can still be kept at a high level under the assumption of limited resources. In fact, the protocol behaves just as in the case of unlimited resources up to the point where more servers than the limit would be needed to keep the success ratio stable. After that point, the protocol behaves like the original ASV protocol (but with the equivalent of a more powerful server) and the success ratio decreases. Nevertheless, it decreases more slowly since now 5, respectively 10, servers handle the incoming requests compared to the single server in the ASV case. Fig. 5(b) shows that the average TTS behaves in a way similar to that of the success ratio. We only checked these properties for an overloading factor of $k = 4$; for higher values of k, the attack level at which stable availability is lost is higher and the rate at which the quality subsequently decreases differs by a constant factor.

5 Related Work and Concluding Remarks

Here we discuss related work on defenses against DoS attacks and their formal analysis. Related work on modular meta-object architectures for distributed systems, and on statistical model checking and quantitative properties has been respectively discussed in Sects. 2.1 and 2.2.

There exist several approaches to formal patterns (see e.g. [10]); ours is different by focusing on executable specifications, quantitative analysis, and the combination of formal and informal aspects. The standard book on security patterns [25] does not discuss DoS defenses, although some of its patterns (such as reflection, replication and filtering) can be related to our patterns.

Defenses against DoS attacks use various mechanisms. An important class of defenses use *currency-based mechanisms*, where a server under attack demands payment from clients in some appropriate “currency” such as actual *money*, *CPU cycles* (e.g., by solving a puzzle), or, as in the case of ASV, *bandwidth*. The earliest bandwidth-based defense proposed was Selective Verification (SV) [14]. Adaptive bandwidth-based defenses include both ASV [15], and the auction-based approach in [28].

Regarding formalizations and analyses of DoS resistance of protocols, a general cost-based framework was proposed in [20]; an information flow characterization of DoS-resistance was presented in the cost-based framework of [16]; and [1] used observation equivalence and a cost-based framework to analyze the availability properties of the JFK protocol. Other works on formal analysis of availability properties use branching-time logics [30,18]. Our own work is part of a recent approach to the formal analysis of DoS resistance using statistical model checking. The first paper in this direction used probabilistic rewrite theories to analyze the DoS-resistance of the SV mechanism when applied to the handshake steps of TCP [3]. ASV itself, applied to client-server systems, was formally specified in rewriting logic and was analyzed this way in [7]. The formalization of ASV in rewriting logic as a meta-object was first presented in [5]. Likewise, cookies have been formalized in rewriting logic as a meta-object for DoS defense in [8].

In this paper we have presented a formal pattern-based approach to the design and mathematical analysis of security mechanisms of Cloud services. We have shown that formal patterns can help deal with security issues and that formal analysis can help evaluate patterns in various contexts. In particular, we have specified dynamic server replication (SR) and the ASV protocol as formal patterns in the executable rewriting logic language Maude. By formally composing the two patterns we have obtained the new pattern $ASV^{+}SR$. We have analyzed properties of the $ASV^{+}SR$ pattern using the statistical model checker PVeStA, and were able to show as our main result that, unlike the two original patterns, $ASV^{+}SR$ achieves stable availability in presence of a large number of attackers at reasonable cost, which can be predictably controlled by the choice of the overloading parameter.

Our current results rely on two simplifications: The client-server communication consists of a stateless request-reply interaction and the replication of servers is only able to add but not to delete servers. As next steps, we plan to refine the patterns to cope with the winding-down of resources at the end of a DoS attack and with more complex client-server interactions where the server has to preserve state. Moreover, in this paper we have only studied quantitative properties of the patterns; it would be very interesting and useful to analyze also qualitative properties. In [8] it is shown that adding cookies to a client-server system preserves all safety properties. We conjecture that the same holds for the ASV and $ASV^{+}SR$ protocols. Finally, we plan to continue with our pattern-based approach and to build a collection of formal patterns for security mechanisms.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just Fast Keying in the Pi Calculus. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 340–354. Springer, Heidelberg (2004)
2. Agha, G., Frolund, S., Panwar, R., Sturman, D.: A linguistic framework for dynamic composition of dependability protocols. IFIP, pp. 345–363 (1993)
3. Agha, G., Gunter, C., Greenwald, M., Khanna, S., Meseguer, J., Sen, K., Thati, P.: Formal modeling and analysis of DoS using probabilistic rewrite theories. In: FCS (2005)
4. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. ENTCS 153(2), 213–239 (2006)
5. AlTurki, M.: Rewriting-based formal modeling, analysis and implementation of real-time distributed services. PhD thesis, University of Illinois (2011)
6. AlTurki, M., Meseguer, J.: PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)
7. AlTurki, M., Meseguer, J., Gunter, C.: Probabilistic modeling and analysis of DoS protection for the ASV protocol. ENTCS 234, 3–18 (2009)
8. Chadha, R., Gunter, C.A., Meseguer, J., Shankesi, R., Viswanathan, M.: Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 39–58. Springer, Heidelberg (2008)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. Dong, J., Alencar, P.S.C., Cowan, D.D., Yang, S.: Composing pattern-based components and verifying correctness. JSS 80, 1755–1769 (2007)
11. Eckhardt, J.: A Formal Analysis of Security Properties in Cloud Computing. Master's thesis, LMU Munich (2011)
12. Erl, T.: SOA Design Patterns. Prentice Hall (2008)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
14. Gunter, C., Khanna, S., Tan, K., Venkatesh, S.: DoS Protection for Reliably Authenticated Broadcast. In: NDSS (2004)
15. Khanna, S., Venkatesh, S., Fatemieh, O., Khan, F., Gunter, C.: Adaptive Selective Verification. In: IEEE INFOCOM, pp. 529–537 (2008)
16. Lafrance, S., Mullins, J.: An Information Flow Method to Detect Denial of Service Vulnerabilities. JUCS 9(11), 1350–1369 (2003)
17. Wirsing, M., et al.: Sensoria Patterns: Augmenting Service Engineering. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 170–190. Springer, Heidelberg (2008)
18. Mahimkar, A., Shmatikov, V.: Game-based Analysis of Denial-of-Service Prevention Protocols. In: IEEE CSFW, pp. 287–301 (2005)
19. MasterCard. MasterCard Statement (September 2011), `http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement`
20. Meadows, C.: A Formal Framework and Evaluation Method for Network Denial of Service. In: IEEE CSFW (1999)

21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. TCS 96(1), 73–155 (1992)
22. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
23. Mühlbauer, T.: Formal Specification and Analysis of Cloud Computing Management. Master's thesis, LMU Munich (2011)
24. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. HOSC 20(1–2), 161–196 (2007)
25. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns. Wiley (2005)
26. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
27. W3C. Request-Response Message Exchange Pattern (September 2011), http://www.w3.org/TR/2003/PR-soap12-part2-20030507/#singlereqrespmep
28. Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D.R., Shenker, S.: DDoS defense by offense. In: ACM SIGCOMM, pp. 303–314 (2006)
29. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. JIC 204(9), 1368–1409 (2006)
30. Yu, C.-F., Gligor, V.: A Specification and Verification Method for Preventing Denial of Service. IEEE T-SE 16(6), 581–592 (1990)