A Language-Based Approach to Prevent DDoS Attacks in Distributed Financial Agent Systems

Elahe Fazeldehkordi, Olaf Owe, and Toktam Ramezanifarkhani

Department of Technology Systems / Department of Informatics, University of Oslo, Norway
{elahefa,olaf,toktamr}@ifi.uio.no

Abstract. Denial of Service (DoS) and Distributed DoS (DDoS) attacks, with even higher severity, are among the major security threats for distributed systems, and in particular in the financial sector where trust is essential.
In this paper, our aim is to develop an additional layer of defense in distributed agent systems to combat such threats. We consider a high-level object-oriented modeling framework for distributed systems, based on the actor model with support of asynchronous and synchronous method interaction and futures, which are sophisticated and popular communication mechanisms applied in many systems today. Our approach uses static detection to identify and prevent potential vulnerabilities caused by asynchronous communication including call-based DoS or DDoS attacks, possibly involving a large number of distributed actors.

Keywords: DoS attacks; DDoS attacks; Active objects; Agent communication; Asynchronous methods; Static analysis; Static detection; Call-based flooding.

1 Introduction

Today distributed and service-oriented systems form critical parts of infrastructures of the modern society, including financial services. In the financial sector security and trust are essential for users of financial services [19]. Security breaches may lead to significant loss of assets, such as physical or virtual money, including cryptocurrencies and bitcoins. In addition, successful attacks on services of a financial institution may damage the trust of customers, which indirectly may hurt the institution [18]. According to [1,12,17], a main threat on financial institutions is Distributed Denial of Service (DDoS) attacks. Protection against DoS/DDoS attacks is therefore crucial for financial institutions. Slow website responses caused by targeted attacks, can imply that customers cannot access their online banking and trading websites during such attacks. Both network layer and application layer DDoS attacks continue to be more and more persistent according to a report from the Global DDoS Threat Landscape Q4 2017. Based on this report, it becomes easier and easier to launch DDoS attacks, and one may even purchase botnet-for-hire services that provide the basis for starting a hazardous DDoS attack. Financial institutions are recommended to monitor the internet traffic to their websites in order to detect and react to possible threats. However, this kind of run-time protection may slow down or temporarily shut down the websites.

Unintended attacks on customers from a financial institution may easily destroy the customer's trust and confidence and result in reputation damage. If customers cannot trust an institution, they may quickly shift to a different institution, due to competition between the many different financial institutions and service providers. Even a single

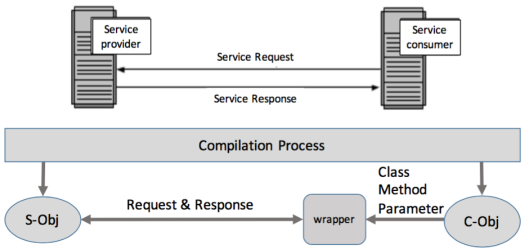

Fig. 1. Distributed communication. S-Obj stands for server object and C-Obj for consumer object.

unfortunate incident of a financial service provider could be enough to influence customers. One should make sure that the software is not harmful for the customers before running it, and this makes static (compile-time) detection more important than in other areas. Thus, in the financial sector static detection is a valuable complement to run-time detection methods, and seems underrepresented.

Call-based *flooding* is commonly seen in the form of application-based DDoS attacks [6]. To prevent DoS/DDoS flooding attacks in a manner complementary to existing approaches, we propose an additional layer of defense, based on language-based security analysis. We focus on DDoS attacks that try to force a (sub)system out of order by flooding applications running on the target system, or by using such applications to drain the resources of their victim.

In this paper, we consider a high-level imperative and object-oriented language for distributed systems, based on the actor model with support of asynchronous and synchronous method interaction. This setting is appealing in that it naturally supports the distribution of autonomous concurrent units, and efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. It is therefore useful as a framework for modeling and analysis of distributed service-oriented systems. Our language supports efficient interaction by features such as asynchronous and non-blocking method calls and first-class futures, which are popular features applied in many distributed systems today. However, these mechanisms make it even easier for an attacker to launch a DDoS attack, because undesirable waiting by the attacker can be avoided with these mechanisms.

We propose an approach consisting of static analysis. We identify and prevent potential vulnerabilities in asynchronous communication that directly or indirectly can cause call-based flooding of agents. More precisely, we adapt a general algorithm for detecting call flooding [14] to the setting of security analysis and for detection of distributed denial of service attacks adding support for many-to-one attacks. The algorithm detects call cycles that might overflow the incoming queues of one or more communicating agents. Each cycle may involve any number of agents, possible involving the attacked agent(s).

The high-level framework considered here is relevant for a large class of programming languages and service-oriented systems.

Outline. Sec. 2 describes the background of the problem. Related work is discussed in Sec. 3. The active object framework is explained in Sec. 4. Our static analysis to prevent attacks is described in Sec. 5. Examples of possible DoS/DDoS attacks are given in Sec. 6. The final section concludes and suggests future work.

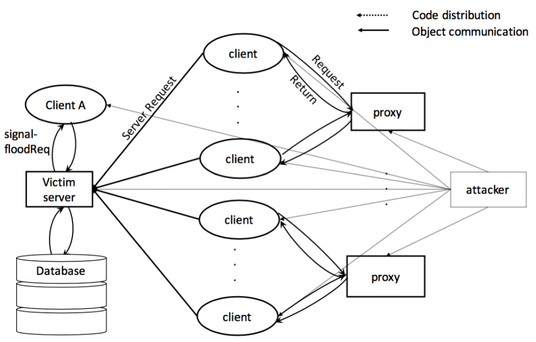

Fig. 2. Distributed object communication in DDoS.

2 Overview

In distributed system communication there is an underlying distributed object system as shown in Fig. 1. In such a distributed system, classes such as server or client classes would be instantiated by objects, and communication is established in the form of method calls, usually wrapped in XML or other forms. Therefore, communication in a distributed system is implemented by method calls between objects. If there is a possibility of flood of requests to the service provider (S-Obj) from the consumer object(s) (C-Obj) in this figure, a DoS attack is probable.

Call-Based Flooding Attacks. To launch a DoS attack, the attacker may try to submerge the target server under many requests to saturate its computing resources. To do so, flooding attacks [6,20] by method calls are effective, especially when the server allocates a lot of resources in response to a single request. Therefore, we detect:

- call-flooding: flooding from one object to another.
- parametric-call-flooding: flooding from one object to another when the target object allocates resources or consumes resources for each call.

In the case of call-flooding, communications are just simple requests like a simple call without parameters or parameters that do not lead to resource consumption. Parametric-call-flooding is when requests usually include parameters in a non-trivial manner. Such requests usually trigger relatively complex processing on the server such as access to a database. Parametric-call-flooding is more effective than call-flooding because it takes fewer requests to drown the target system. However, call-flooding are more common and easier for attackers to exploit.

Categories of Call-Based Flooding Attacks: DoS or DDoS possibilities.

one-to-one (OTO): If thousands of requests every single second come from one source object to a target object, then it is a *one-to-one (OTO) DoS attack*. The intent of the flooding might be malicious, or even undeliberate call cycles. Communication between Client A and the victim server in Fig. 2 is an example of this attack.

many-to-one (MTO): If the incoming flooding traffic originates from many distinguishable different sources, then it is a *many-to-one (MTO) DDoS attack*. Fig. 2 shows a distribution of code between clients and a server, and proxies.

one-to-many (OTM): A *one-to-many attack* appears if a system makes an unlimited number of requests to many objects simultaneously. Such an attack can be serious since many target objects are attacked at the same time.

Static Attack Detection and Prevention. For any set of methods that call the same target method, a call cycle could be harmful. The methods might belong to the same or different objects with the same or different interfaces. In the case of normal blocking calls, where the caller is blocking while waiting for the response, making a flood of requests also means receiving a flood of responses. And thus in the case of OTO, it may cause a self DoS for the attacker. With the possibility of non-blocking calls in a distributed setting, it is more cost-beneficial for an OTO attacker to launch a DoS, because then undesirable blocking by the attacker is avoided. By means of futures and asynchronous calls, a caller process can make non-blocking method calls.

The possibility of unbounded object creation, referred to as *instantiation flooding* [8], could cause resource consumption and DoS that could be detected statically, especially if those objects and their communication can cause flooding requests from the bots, such as the customers in our example. It is even worse if there is instantiation flooding on the target side of the distributed code. This can be detected by static analysis of the target. (See the example in Fig. 10.) Our static analysis detects explicit or implicit call-flooding. Static detection is accomplished by static analysis at compile time and informs the programmer about the possibility of program exploitation at runtime.

3 Related work

A DoS attack, or its distributed version, happens when access to a computer or network resource is intentionally blocked. Considering the exploited vulnerabilities, these attacks might be classified by resource consumption attacks or flooding attacks, of which the latter category is the most common [6]. In this paper, we aim to prevent distributed code to be exploited by attackers to launch a DoS attack by detection of possible call-based flooding in both of the target and zombie sides. To do so, we analyze the distributed code to make an additional layer of defense against DoS or DDoS attacks.

In the following, we discuss related works for preventing application-based DDOS attacks using static detection. In the paper presented by Chang et al. [3], a novel static analysis approach was introduced in order to detect semantic vulnerabilities in networked software that might cause denial of service attacks because of resource exhaustion. Their approach is implemented in a tool named SAFER: Static Analysis Framework for Exhaustion of Resources. SAFER integrates taint analysis (in order to compute the group of program values that are data-dependent on network inputs) and control dependency analysis (for computing the group of program statements whose execution can affect the execution of a given statement) toward detecting high complexity control structures that can be caused by untrusted network inputs. The tool applies the CIL static analysis framework and combines different heuristics for recognizing loops and recursive calls. Compared to our work the SAFER approach is oriented toward detecting server attacks from within the server code, whereas our approach is mainly targeting server attacks from an external attacker, or a combination of external agents. An attacker needs to understand the code of the server in order to find weaknesses that can be triggered by specific inputs. In contrast, our approach is detecting attacks caused by coordination of several agents and/or servers in a distributed setting.

Another work that detects resource attacks from within the server code is presented by Qie et al. [15]. In their toolkit, they check for possible “rule” violations at runtime.

This work is complementary to ours, since our work is oriented toward static detection. Gulavani and Gulwani [7] describe a precise numerical abstract domain. This domain can be used to prove the termination of a large class of programs and also to estimate valuable information such as timing bounds. In order to make linear numerical abstract domains more precise, they make use of two domain lifting operations: One operation depends on the principle of *expression abstraction*. This describes a set of expressions and determines their semantics by use of a selection of directed inference rules. It works by picking up an abstract domain and a group of expressions, such that their semantics are described by a group of rewrite rules, in order to construct a more precise abstract domain. The second domain constructor operation picks up a linear arithmetic abstract domain and constructs a new arithmetic domain that is able to represent linear relations through introduction of *max* expressions. Another approach to estimate worst-case complexity is presented by Colon and Sipma [4]. These approaches [7,4], in which the complexity of loops and recursive calls has been estimated using structural analysis, are widely complementary to our work.

Zheng and Myers [21] propose a framework for using static information flow analysis in order to specify and enforce end-to-end availability policies in programs. They extend the decentralized label model to include security policies for availability. This work presents a simple language with fine-grained information security policies described by type annotations. In addition, this language has a security type system to reason about end-to-end availability policies. Various examples have been discussed, in which abuse of an availability policy can represent denial of service attacks.

In a work by Meadows [13], a formal analysis has been developed in order to apply the maximum benefit of tools and approaches that have already been used to strengthen protocols against denial of service attack. This analysis has been done at the protocol specification level. Also, different ways in which existing cryptographic protocol analysis tools can be modified for the purpose of operating in this formal framework, have been demonstrated. In contrast, we do a detailed static analysis of source code both inside and outside a server. The class of software vulnerabilities that we can detect is more complicated than what appears just at the network-protocol specifications level. Moreover, vulnerable sections of the source code have been identified in our work.

The current work shows how the static analysis method for detection of flooding can be used for detection of DoS and DDoS attacks. This general idea was also outlined by the same authors in an extended workshop abstract [16]. Moreover we here discuss why this is particularly harmful in the financial sector, where both economic assets and customer trust are at risk. Furthermore we simplify and adapt the static analysis method of [14] to the setting of financial service systems, extending it to detect many-to-one attacks involving unbounded creation of objects (as demonstrated in example 10), as well as hidden attacks, neither of which were detected by the original method of [14].

4 Our Framework for Active Object Systems

The setting of concurrent objects communicating by asynchronous method calls combines the Actor model and object-orientation, and is referred to as *active objects*. Active object languages are suitable for modeling and implementing distributed applications, letting a distributed system be modeled by a number of active objects that interact via asynchronous method calls. The active object model provides natural description of autonomous agents in a distributed system, and the *future mechanism* provides an efficient communication primitives [2,11], allowing results computed in a distributed setting to be referred to and shared. Moreover, the addition of *cooperative scheduling*, as suggested in the Creol language [10], allows further communication efficiency, by adding process scheduling control in the programming language, and passive waiting. This is achieved by including statements for suspension control, and letting each object have a process queue for holding suspended processes. We consider a core language for active objects with future-based communication primitives, inspired by Creol and ABS [9]. The objects are concurrent units distributed over a network, and their identity is globally unique. An object has a process queue, as well as a queue for incoming method call requests, and can perform at most one process (i.e., remaining part of method call) at a time. A process can be suspended by an **await** statement, allowing other (enabled) processes to continue. When a process is ended or suspended, the object may continue with an incoming call request or other enabled process from the process queue (if any). The **await** statement allows a process to wait for a Boolean condition to be satisfied, or for a future value to be available. The statement is enabled when the waiting condition/future is satisfied/available. The **await** statement enables high-level process control, instead of low-level process synchronization statements such as signaling and lock operations.

Our core language is a typed, imperative language. An assignment has the form $x := e$ where the expression e is without side-effects. All object variables (i.e., object references) are typed by an interface, and an interface specifies the set of methods that are visible through that interface. The interfaces of a class protect and limit the object communication, and in particular shared variable interaction is forbidden. Local data structure is made by data type declarations, indicated by **data**, and a functional data type sublanguage is used to create and manipulate data values. Data values are passed by value, while object variables are passed by reference. The language supports first-class futures. The basic interaction mechanisms (by method calls/futures) are as follows:

- $f := o!m(\overline{e})$ – the current object calls method m on object o with actual parameters $\overline{e}$. A globally unique identity u identifying the call is assigned to the future variable f. A message is then sent over the network from the current object to object o. When object o eventually performs the method and the method gives a result defined by a **return** statement, that result is placed in a (globally accessible) future with identity u, and the future u is then said to be *resolved*. Any process of any object that knows u may access the future value or wait for it to be available.
- $x :=$ **get** f – this statement blocks until the value of the future f is available, and then that value is assigned to the variable x. (Here f may be an expression resulting in a future identity.)
- **await** c – this statement suspends if the Boolean condition c is not satisfied, and is enabled when c is satisfied,
- **await** $x :=$ **get** f – this statement suspends if the value of f is not yet available, and is enabled when the future is available. Then the future value is assigned to x.

The statement sequence $f{:=}\ o!m(\overline{e}); x{:=}$**get** f corresponds to a traditional blocking call, and is abbreviated $x{:=}\ o.m(\overline{e})$ using the conventional dot-notation. The statement sequence $f := o!m(\overline{e})$; **await** $f; x :=$ **get** f is abbreviated **await** $x := o.m(\overline{e})$ and corresponds to a non-blocking call, since the **await**-statement ensures that the future is available before the **get**-statement is performed. If the result value is not needed, we may simplify the syntax to $o.m(\overline{e})$ for blocking calls and **await** $o.m(\overline{e})$ for non-blocking calls. And if the future is not needed, $f := o!m(\overline{e})$ may be abbreviated to $o!m(\overline{e})$, in which case the future cannot be accessed (since it is not stored in a future variable).

Object creation has the syntax $x :=$ **new** $C(\overline{e})$ **at** o, where the class parameters behave like fields (initialized to the values of $\overline{e}$) except that they are read-only, and the new object is created locally at the site of object o. With the syntax $x :=$ **new** $C(\overline{e})$ the new object is located anywhere in the distributed system.

One may refer to the current object by this and to the caller object by (the implicit method parameter) caller. Self calls are possible by making calls to this, and recursion is allowed. *Active behavior* is possible by making a recursive self call in the constructor method (given as a nameless method). By means of suspension, the active self behavior may be interleaved with execution of incoming calls from other objects, thereby combining active behavior and passive behavior. If- and while-statements are as usual.

We assume all class parameters and method parameters (including this and caller) are read-only. This helps the static analysis by reducing the set of false positives. For methods that return no information we use a predefined type $Void$ with only one value, $void$. For simplicity we omit **return** $void$ at the end of $Void$ methods in the examples.

5 Static Analysis to Prevent Attacks

We base our approach on the static analysis of flooding presented in [14] for detection of flooding of requests, formalized for the Creol/ABS setting with futures. We adapt this notion of flooding to deal with detection of DDoS attacks, which have a similar nature. The static analysis will search for flooding cycles in the code, possibly involving several classes. According to [14] (unbounded) flooding is defined as follows:

Definition 1 (Flooding). *An execution is* flooding with respect to a method m *if there is an execution cycle C containing a call statement to a method m at a given program location, such that this statement may produce an unbounded number of uncompleted calls to method m, in which case we say that the call is* flooding with respect to C *in the given execution.*

Like in [14], we distinguish between weak flooding and strong flooding. Strong flooding is flooding under the assumption of so-called *favorable* process scheduling, i.e., enabled processes are executed in a fair manner.

Definition 2 (Strong and weak flooding). *A call is* weakly flooding *with respect to a cycle C if there is an execution where the call is flooding with respect to C. And a call is* strongly flooding *with respect to a cycle C if there is an execution with fair scheduling of enabled processes where the call is flooding with respect to C.*

Strong flooding reflects the more serious flooding situations that persist regardless of the underlying scheduling policy. In the detection of strong flooding, a *statically enabled* node is considered strongly reachable if each of its predecessor flow nodes are strongly reachable. All statements are statically enabled, apart from **get**/**await**

1. Make separate *control flow graphs* (CFGs) for each method. Include a node for each *call*, *get*, *await*, *new* (for object creation), *if* and *while* statement, as well as an initial *starting* and a final *return* node.
2. Add call edges from call nodes to the start node of a *copy* of the called method. In case the call is recursive, simply add a call edge to the existing start node.
3. Identify any cycles in the resulting graph (including all copies of the CFGs).
4. Assign a unique label to each call node, and assign this label to the *start* and *return* node of the corresponding copy of the method CFG.
5. Make *put* edges from the *return* nodes to the corresponding *get/await* nodes. This requires static flow analysis, possibly with over-approximation of *put* edges.

Fig. 3. Control Flow Graph.

Consider a cycle C in the control flow graph G resulting from Fig. 3:

1. Mark all nodes in C as*strongly-reachable (SR)*, and the rest as (initially) *not reachable*.
2. From the entry point to the cycle, follow all flow and call edges in a depth-first traversal of G and mark the nodes as *weakly-reachable (WR)*, *strongly-reachable (SR)*, or neither, as defined in Def. 3.
3. If the previous step results in any changes to the *SR* or *WR* node sets, go to step 2.
4. Report flooding of call n if $n \in (calls - comps)$ where $calls = \{n \mid call_n \in WR\}$ and $comps = \{n \mid return_n \in SR \vee get_n \in SR\}$.

Fig. 4. Algorithm for detecting flooding by means of $calls$ and $comps$ sets in a given cycle.

statements. A **`get`** statement or an **`await`** on a future/call is statically enabled if the corresponding future/result is available, detected statically if the corresponding return statement is strongly reachable or another **`get/await`** statement on the same future is strongly reachable. We rely on a static under-detection of the correspondence between return statements and futures. In the examples this detection is straight forward. With respect to DDoS, weak flooding of a server is in general harmless unless the flooding is caused by a large enough number of objects. Strong flooding is dangerous even from a single attacker.

Following [14], flooding is detected by building the control flow graph (CFG) of the program, locating control flow cycles as outlined in Fig. 3, and then analyzing the sets of weakly reachable calls, denoted *calls*, and the set of strongly reachable call completions, denoted *comps*, in each cycle. Flooding is reported for each cycle with a nonempty difference between *calls* and *comps*, as explained in Fig. 4. Note that the abbreviated notations for synchronous calls and suspending calls are expanded to the more basic call primitives, as explained above. We assume that assignments (other than calls) will terminate efficiently and therefore ignore them in the CFG. For each method the CFG begins with a start node and ends with a return node (even for void methods) – the latter helps in the analysis of method completion. We next define *weakly and strongly reachable nodes*. The detection of strongly reachable nodes uses a combination of forward and backward analysis, and is simplified compared to [14]:

Definition 3 (Weakly and strongly reachable nodes). *Consider a given cycle C.* Weakly reachable (WR) nodes *are those that are on the cycle or reachable from the cycle by following a flow edge or a call edge.*

A node is strongly reachable (SR) *if it is in the given cycle or is reachable from an SR node without entering an if/while node nor passing a wait node (**`get/await`**)* *outside the cycle, unless the return node of the corresponding call is strongly reachable. A return node is SR if there is a SR* **`get/await`** *node on the same future. And a node is SR if all its predecessor nodes are SR.*

We consider two versions of SR, the optimistic*, where we follow call edges (as indicated above), and the* pessimistic*, where we follow a call edge n only when the call is known to complete, i.e., when $n \in SR$ before following the call edge. (As above we follow flow and put edges without restrictions.)*

The optimistic version is used to find unbounded flooding under the assumption of favorable scheduling, i.e., *strong flooding*. The pessimistic version is used to detect unbounded flooding without this assumption, i.e., *weak flooding*. Detection of strong flooding implies detection of weak flooding, but with less precise details about which calls that possibly may cause flooding. If there is a call causing weak flooding wrt. a given cycle, pessimistic detection will report this call or a call leading to this call. If there is a call causing strong flooding for a given cycle C, optimistic detection will report this call. Our notions of optimistic and pessimistic reachability cover a wider class of nodes than in [14]. The soundness of [14] can be generalized to our setting.

6 Examples of Possible DoS/DDoS Attacks

An Example of Flooding Cycles. We consider here an example of a possible DoS attack on customers caused by a financial institution. The attack may be unintended by the institution, but may result from an update supposed to give better efficiency, by use of the future mechanism to reduce the amount of data communicated over the network.

We imagine that the financial institution has a subscription *service* for customers, such that customers can register and receive the latest information about shares and funds, through data of type “newsletter”, here simply defined as a product type consisting of a *content* and a *date*. The financial institution uses a method signal to notify the customers about new information about shares. In the first version, each call to signal has the newsletter as a parameter. This may result in heavy network traffic and many of the newsletters may not be read by the customers. In the “improved” solution, each call to signal contains a reference (by means of a future) to the newsletter rather than the newsletter itself. However, this allows the subscription service system to send signal calls even before the newsletter is available, and as we will see, this can cause a DoS attack on the subscribing customers.

In order to handle many customers, a (dynamic) number of proxies are used by the service object, and an underlying newsletter producer is used for the sake of getting newsletters, using suspension when waiting for news. The proxies are organized in a list (*myCustomers*), growing upon need. In both solutions, futures are used by the service object to avoid delays while waiting for a newsletter to be available. In this way the service object can continuously respond to customers. The interfaces are shown in Fig. 5. We abbreviate “Newsletter” to “News”. Fig. 6 represents a high-level implementation of the publish/subscribe model, adapted from [5]. A multi-cast to each object in the *myCustomers* list is made by the statement *myCustomers!signal(ns)* in line 13. If we shift requiring the actual newsletter to have arrived, from the Proxy (as shown by the statements *ns:=get fut; myCustomers!signal(ns)* in the original *publish* method) to the Customer (i.e., *news:=***`get`** *fut* in the modified *Customer.signal* method). This change

```
1      data News == (String content, Int date) // a product data type
2      interface ServiceI{
3              Void subscribe(CustomerI cl)  // called by Clients
4              Void produce()}  // called by Proxies
5      interface ProxyI{
6              ProxyI add(CustomerI cl) // called by Service
7              Void publish(Fut[News]fut)} // called by Service
8      interface ProducerI{
9              News detectNews()} // called by Service
10     interface NewsProducerI{
11             Void add(News ns) // called when news arrives
12             News getNews()} // called by Producers
13     interface CustomerI{
14             Void signal(News ns)} // called by Proxies
```

Fig. 5. The interfaces of the units in the subscription example.

```
1    class Service(Int limit, NewsProducerI np) implements ServiceI{
2       ProducerI prod; ProxyI proxy; ProxyI lastProxy; //declaration of fields
3       { prod := new Producer(np); proxy:= new Proxy(limit,this); lastProxy:=proxy; this!produce()}
4       Void subscribe(CustomerI cl) {lastProxy:=lastProxy.add(cl)}
5       Void produce(){ Fut[News]fut :=prod!detectNews(); proxy!publish(fut)}} // sends future
6    class Proxy(Int limit,ServiceI s) implements ProxyI{
7       ProxyI nextProxy; List[CustomerI] myCustomers:=empty; //fields
8       ProxyI add(CustomerI cl){ ProxyI lastProxy:=this;
9         if length(myCustomers)<limit then myCustomers:=append(myCustomers,cl)
10        else if nextProxy=null then nextProxy:= new Proxy(limit,s) fi;
11        lastProxy:=nextProxy.add(cl) fi; return lastProxy}
12      Void publish(Fut[News]fut){ News ns :=get fut; // wait for the future
13        myCustomers!signal(ns); // multi-cast the result
14        if nextProxy=null then s!produce() else nextProxy!publish(fut) fi}}
15   class Producer(NewsProducerI np)implements ProducerI{ // Wrapper for the news producer:
16      News detectNews(){ News news; news:=np.getNews(); return news}}
17   class NewsProducer()implements NewsProducerI{ List[News] nl;
18      Void add(News ns){nl:=append(nl,ns)}
19      News getNews(){News n; await nl /= empty; n:=first(nl); nl:=rest(nl); return n} }
20   class Customer implements CustomerI{ // Consumer of news items:
21      News news; // the latest news
22      Void signal(News ns){news:=ns}}
```

Fig. 6. Classes providing an implementation of the subscription example.

```
1    class Proxy(Int limit,ServiceI s) implements ProxyI{
2       ProxyI nextProxy; List[CustomerI] myCustomers:=empty;
3       ProxyI add(CustomerI cl){ ... }
4       Void publish(Fut[News]fut){myCustomers!signal(fut); // send future, no waiting
5         if nextProxy=null then s!produce() else nextProxy!publish(fut) fi}}
6
7    class Customer implements CustomerI{ News news; ...
8       Void signal(Fut[News] fut) { news:=get fut}}  // blocking wait
```

Fig. 7. DoS attack by a variation of the subscription example.

in the program causes flooding of customers.

Service.produce asynchronously calls Producer.detectNews (Pd), line 5 of Fig. 6
Service.produce asynchronously calls Proxy.publish (Xb), line 5 of Fig. 6
Proxy.publish asynchronously calls Customer.signal (Cs), line 6 of Fig. 7
Proxy.publish asynchronously calls Service.produce (Sp) line 6 of Fig. 7

Each iteration of this cycle generates an asynchronous call to *Proxy.publish*, which again produces an asynchronous call to *Producer.detectNews*, which is not processed as part of this cycle, nor is its processing synchronized call with the cycle. An unbounded number of suspended calls to *Producer.detectNews* can be produced by this cycle. We then say that the cycle is flooding. The flooding cycle identified above is harmless pro-

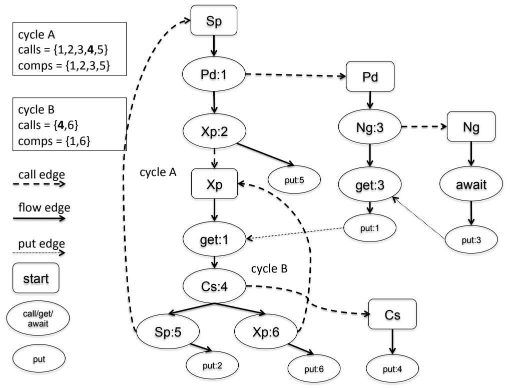

Fig. 8. The graph and call/comp sets for the original version of the program (Fig. 6).

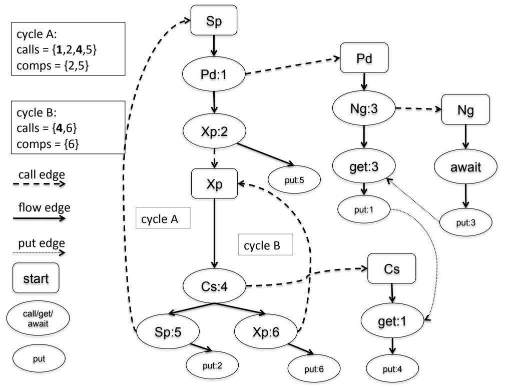

Fig. 9. The graph and call/comp sets for the modified version of the program (Fig. 7).

vided the customers are able to process their signal calls as fast as the cycle iterations. The programmer will be warned by our algorithm about each possible flooding, and should determine whether it is a real problem.

In contrast, the modified program version (Fig. 7) does not wait in *Proxy.publish* (doing *ns*:=**get** *fut*) until the newsletter is produced. Instead the future is directly passed to another asynchronous call $(myCustomers!signal(fut))$ in line 6 of Fig. 7 through the method *Proxy.publish*, this removes any progress dependency between the cycle producing the *Producer.detectNews* and *Customer.signal* calls and the processing of those calls. The completion of the *Producer.detectNews* and *Customer.signal* calls does not only depend on the speed of code execution, but depend on the rate of newsletter items arrivals. Practically, this flooding cycle generates a number of unprocessed calls that quickly grows to system limits.

Applying the Algorithm to the Example. Following [14], the *call* and *comps* sets for the two publish/subscribe versions are shown in Figures 8 and 9. Method names are abbreviated with two letters as indicated above, letting *Ng* abbreviate method *getNews* of interface *NewsProducerI*. There are two cycles in Fig. 8, i.e., cycle A and B. We have

```
class Attacker(ServerI s) {
  {this!run(); } // initialization
  Void run() { ClientI c := new Client(); c!connect(s); this!run() }//terminate& make recursive call
}
class Client() implements ClientI{
  Nat connect(ServerI s){
    Nat n := s.register(); return n }// blocking call, so each client will not cause flooding.
}
class Service(DataBase db) implements ServiceI{ {...} // Initialization
  Nat register(){Nat n :=0; if okcheck(caller) then Bool ok := db.open();
      if ok then n:=db.add(caller); db.query(...); db.close() fi fi; return n }
//register requires time and resources
  ... }
```

Fig. 10. Flooding by unbounded creation of innocent clients targeting the same server.

a flooding on the call to *Customer.signal* (Cs) in both cycles. However, this flooding does not reflect an actual flooding since the Customer objects easily keep up with the calls since the amount of work required by the Customer to complete a signal call is trivial. The execution rate is restricted with respect to the actual arrival of new items from the *NewsProducer* (by the blocking call in the proxies), and therefore, the rate of produced asynchronous calls to *Customer.signal* by this cycle is limited. Thus this is an example of *weak flooding* that is harmless. Furthermore, cycle *B* is not infinite since it goes through the chain of Proxies. The modified version of the program is shown in Fig. 9. This version is displaying *strong flooding*. The flooding-cycle of *Pd* (Producer.detectNews) through both cycles is dangerous and will cause flooding of the system instantly. In version 1, there is a *get* in cycle A that regulates the speed of this cycle, whereas in the modified version there is no *get* in cycle A.

An Example of Instantiation Flooding. The example in Fig. 10 shows how a *Client-Distribution* object can cause an attack by using an unbounded number of clients to flood the same server s, due to an unbounded recursion of the *run* method. The initialization of the attacker object of class ClientDistribution connects to a client, and the client do the registration of the server object. The attacker may start such a communication with lots of clients to register at the same server. (For simplicity, interfaces are omitted here.) Each client is innocent in the sense that it does not cause any attack by itself. By finding such a vulnerability in the ClientDistribution, an attacker can cause the flooding attack by calling run(). In addition, the non-blocking call in this method helps the attacker because the method does not wait for the connect calls to complete, therefore it is able to create more and more workload for the server s in almost no time. The execution of f:=c!connect(s) causes an asynchronous call and assigns a future to the call. Thus no waiting is involved. The run method recursively creates more and more objects, located somewhere in the distributed network. Therefore, the attacker creates flooding by rapidly creating clients that each performs a resource-demanding operation on the same server. Static analysis detects such attacks by finding a call loop (in this case inside run) which is also targeting the same server.

In this example, if the object creation in run had happened locally, an explicit instantiation flooding that consumes all the resources in an object will happen, which is a self DoS attack. However, since the object creation is distributed, the example in Fig. 10 shows an implicit attack because of targeting the same server by different clients.

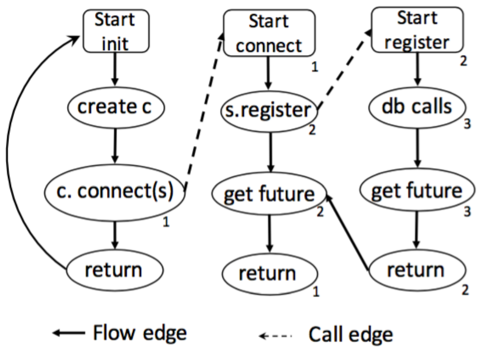

Fig. 11. Static detection of flooding using unbounded creation.

Static Analysis of the Instantiation Example. Consider the example in Fig. 10. For the run method of class ClientDistribution, the following cycle is detected:

the initialization of the attacker calls *run*
run creates a client object *c*
run calls c!connect(s)
run terminates and calls itself recursively in an asynchronous call.

The *run* call has a call edge to the flow graph of *connect* (call 1), and *connect* has a call edge to the flow graph of *register* (call 2). The call to *register* waits for completion of *register* since it is a blocking call, and the database calls (call 3) made by *register* wait for the completion of these database calls. The code for the database is not given, and therefore the analysis will be worst-case by considering the termination of such calls non-reachable (unless indirectly found strongly reachable). The control flow graph is given in Fig. 11. The set of weakly reachable call nodes of the cycle, i.e., *calls*, are $\{1, 2, 3\}$ with optimistic detection, and $\{1\}$ with pessimistic detection. And the set of strongly reachable calls, i.e., *comps*, is empty in both cases. This gives that the set of potentially flooding calls, given by *calls* − *comps*, is $\{1\}$ (c.connect) with pessimistic detection and $\{1, 2, 3\}$ with optimistic detection. However, in this case, call 1 does not reflect a real flooding since each call is on a separate object, but call 2 (s.register) and call 3 (the db calls) do. We detect strong flooding. The example may be improved by using suspending calls (using **`await`**) on the database operations.

6.1 Modification of the Static Detection to DDoS

As seen in Fig. 2, a DDoS attack on a server is often made through many innocent clients. This is hard to detect from the server side at runtime since each client may behave in an acceptable manner, and since the real attacker is hidden behind the clients. The original detection method [14] is not oriented towards such attacks, since it is not aware of the number of generated objects of a class. Moreover, the approach is using as an assumption that *an execution has a bounded number of objects*. Nevertheless, if applied to the example in Fig. 10, it will report a possible attack on the clients (treating all customers as one object), but not an attack on the Service server, which is the real attack. A draw-back is that there could be reported more false positives due to overapproximation.

A weakness with pessimistic detection is that the *connect* call, but not the *register* call, would be reported. Although the former call leads to the second, the detection result is not appropriate since a harmless call is reported an not the harmful one. Another weakness is that the attack would not be discovered when removing the connect call from the attacker class and instead letting the register call be caused by the init method of class Client. (In this case the method parameter s should be transferred as a class parameter.) The reason for this is that indirect calls due to object generation are ignored (since by assumption there cannot be unboundedly many such calls). To compensate these weaknesses, we make two modifications wrt. [14], described below:

First, we modify the static analysis by viewing a **new** C statement as a special kind of a call statement with its own associated call number and a call edge to a copy of the *init* code of class C, which again may have further calls, treated as usual. More precisely, we treat **new** C as a *simple* call statement (like $new!C(classparameters)$ except that the new object is not known before the call) since the **new** statement does not wait for the *init* to complete. This allows us to see the generation of objects and to follow all implicit calls from the initialization code. Thus we can detect instantiation flooding attacks depending on call indirectly caused by object initialization. We may assume that an initialization cannot generate flooding in itself since each initialization is on a new object. Thus the call numbers associated with object creation can be included in *comps*. Furthermore, one more call on a new object cannot generate flooding on this object (unless in a cycle after the object creation). The same goes for a finite number of calls on a new object, if it can be detected statically that all these calls have the same new object as callee. These calls can also be included in *comps*. In the example of Fig. 10, we detect that the call c!connect is to the new Client object, and this call will then not be reported with the improved static detection.

Secondly, since implicit calls are important in DDoS attacks, we follow all call edges in the calculation of WR nodes, even in the pessimistic version. The resulting improved static detection method can then also detect the hidden attacker in all versions of the instantiation example, as shown below. Since the improved static detection method depends on static detection of same callee, we briefly discuss how to incorporate this: Two calls in the same method activation have the same callee if the callee is the same variable, and it is either

– a read-only variable (such as a parameter),
– a local variable and there are no updates on this variable between the calls, or
– a field variable and there are no updates on it nor suspension between the calls.

The first of these calls may be an object creation $x :=$ **new** $C(\ldots)$, and the second a call with x as callee (provided one of the conditions above are satisfied for x). This suffices for the example with the two calls $c :=$ **new** $Client()$ and $f := c.connect(s)$. This detection could be improved in several ways. In particular, we may detect that the actual parameter s in the latter call refers to the same object for all activations of *run* since s is a read-only class parameter and the recursive *run* call is on the same object (since this is read-only). This could be used in the detection algorithm to see that all *s.register* calls refer to the same server s, which gives a clear indication of a DDoS attack.

Instantiation example revisited. We reconsider the example in Fig. 10, using the improved static detection algorithm. Now the *create c* node of Fig. 11 is represented as a call, say call 0. For the original version of the example, the initialization is empty so call 0 is considered terminating ($0 \in comps$). We get $calls = \{0, 1, 2, 3\}$ where 0 corresponds to the creation of the new C object. But call 0 (c:= new Client) and call 1 (c!connect) do not generate flooding since they are on a new object. Thus $comps = \{0, 1\}$. This shows that there is a possibility of call flooding through call 2 (s.register) and call 3 (the db calls); and these correspond to actual attacks. For the modified version of Fig. 10, where the register call is caused by the Client initialization, we get a similar analysis except that there is no call 1 (c!connect) since this is incorporated in the Client initialization. Thus we get that $calls = \{0, 2, 3\}$ and $comps = \{0\}$. Here the presence of call 0 enables us to detect call 2 in the (modified) initialization code and thereby also call 3 (and both correspond to possible flooding).

7 Conclusion

In this paper we have considered denial of service attacks, formulated in a high-level imperative language based on concurrent objects communicating by asynchronous calls and futures, thereby supporting asynchronous as well as synchronous communication. The language includes mechanisms for process control allowing non-trivial process synchronization by means of cooperative scheduling. We adapt a static detection algorithm developed for analysis of flooding to this setting, in order to detect possible denial of service attacks. This kind of static analysis is useful in the financial sector, because the aspect of trust between customers and service providers is essential, perhaps more so than in other application areas, and therefore static detection is valuable.

We have illustrated the approach on examples of distributed systems in the financial sector, including versions of a one-to-many attack and a many-to-one attack. In the first example a financial institution notifies a number of subscribing customers. We have seen that a revision of the basic notification software used by the financial institution, intended to be more efficient, actually implies a one-to-many attack on the subscribing customers. In this example, the financial institution was responsible for the attack, which could lead to loss of reputation and of customers. Static detection solved the situation here since the detection is made before the program is run. The underlying detection algorithm is sound for call-based coordinated attacks, provided the source code of the objects involved in the coordinated attack is available. In the many-to-one example, an attacker object causes an attack by using an unbounded number of clients, each innocent, to flood the same server s, letting a new client be created in each cycle.

In this paper we have adapted a general algorithm for detecting flooding [14] to the setting of DDoS and improved it to deal with unbounded object generation and to better reveal hidden attacks. Our framework can deal with advanced programming mechanisms including suspension and first-class futures considering distributed systems at a high-level of abstraction. It is therefore relevant for high-level modeling and prototyping of distributed software solutions. In future work, we suggest to complement the static checking with dynamic runtime checking since static detection methods give a degree of over-estimation. This could give a more precise combined detection strategy.

Acknowledgments. We thank the reviewers for significant feedback. This work is supported by the *IoTSec* project, the Norwegian Research Council (No. 248113/O70), and by the *SCOTT* project, the European Leadership Joint Undertaking under EU H2020 (No. 737422).

References

1. Ashford, W.: DDoS is most common cyber attack on financial institutions (blog post, 2016), https://www.computerweekly.com/news/4500272230/DDoS-is-most-common-cyber-attack-on-financial-institutions/
2. Boer, F.D., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., et al.: A survey of active object languages. ACM Computing Surveys (CSUR) **50**(5), 76 (2017)
3. Chang, R., Jiang, G., Ivancic, F., Sankaranarayanany, S., Shmatikov, V.: Inputs of Coma: Static detection of denial-of-service vulnerabilities. In: 22nd IEEE Computer Security Foundations Symposium (CSF'09). pp. 186–199. IEEE Computer Society (2009)
4. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 67–81. TACAS 2001, Springer-Verlag, London, UK, UK (2001)
5. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Logical and Alg. Methods in Prog. **83**(5), 360 – 383 (2014)
6. Douligeris, C., Mitrokotsa, A.: DDoS attacks and defense mechanisms: Classification and state-of-the-art. Computer Networks **44**(5), 643–666 (2004)
7. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: International Conference on Computer Aided Verification. pp. 370–384. Springer (2008)
8. Jensen, M., Gruschka, N., Herkenhöner, R.: A survey of attacks on web services. Computer Science-Research and Development **24**(4), 185 (2009)
9. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. pp. 142–164. Springer (2012)
10. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software & Systems Modeling **6**(1), 35–58 (Mar 2007)
11. Karami, F., Owe, O., Ramezanifarkhani, T.: An evaluation of interaction paradigms for active objects. J. Logical and Alg. Methods in Prog. **103**, 154 – 183 (2019)
12. Lambert, K.: Protecting financial institutions from DDoS attacks (blog post, 2018), https://www.imperva.com/blog/protecting-financial-institutions-from-ddos-attacks/
13. Meadows, C.: A formal framework and evaluation method for network denial of service. Computer Security Foundations Workshop. Proceedings of the 12th IEEE pp. 4–13 (1999)
14. Owe, O., McDowell, C.: On detecting over-eager concurrency in asynchronously communicating concurrent object systems. J. Logical and Alg. Methods in Prog. **90**, 158 – 175 (2017)
15. Qie, X., Pang, R., Peterson, L.: Defensive programming: Using an annotation toolkit to build DoS-resistant software. CM SIGOPS Operating Systems Review, 36(SI) pp. 45–60 (2002)
16. Ramezanifarkhani, T., Fazeldehkordi, E., Owe, O.: A language-based approach to prevent DDoS attacks in distributed object systems. In: 29th Nordic Workshop on Programming Theory. Turku Centre for Computer Science (Nov, 2017), (extended abstract, 3 pages)
17. Urrico, R.: DoS services: Verisign (2018), https://www.cutimes.com/2018/07/03/denial-of-service-attacks-overwhelmingly-target-fi/?slreturn=20190713065814/
18. Wilczek, M.: Why banks shouldn't be in denial about DDoS attacks (blog post, 2018), https://www.globalbankingandfinance.com/why-banks-shouldnt-be-in-denial-about-ddos-attacks/
19. Zahoor, Z., Ud-din, M., Sunami, K.: Challenges in privacy and security in banking sector and related countermeasures. Intern. Computer Applications **144**(3), 24–35 (2016)
20. Zargar, S.T., Joshi, J., Tipper, D.: A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. IEEE Comm. Surveys Tutorials **15**(4), 2046–2069 (2013)
21. Zheng, L., Myers, A.C.: End-to-end availability policies and noninterference. Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop pp. 272–286 (2005)