Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts

Zhang Fu, Marina Papatriantafilou, Philippas Tsigas
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg Sweden
{zhafu,ptrianta,tsigas}@chalmers.se

Abstract

A weak point in network-based applications is that they commonly open some known communication port(s), making themselves targets for denial of service (DoS) attacks. Considering adversaries that can eavesdrop and launch directed DoS attacks to the applications' open ports, solutions based on pseudo-random port-hopping have been suggested. As port-hopping needs that the communicating parties hop in a synchronized manner, these solutions suggest acknowledgment-based protocols between a client-server pair or assume the presence of synchronized clocks. Acknowledgments, if lost, can cause a port to be open for a longer time and thus be vulnerable to DoS attacks; Time servers for synchronizing clocks can become targets to DoS attack themselves.

Here we study the case where the communicating parties have clocks with rate drift, which is common in networking. We propose an algorithm, BIGWHEEL, *for servers to communicate with multiple clients in a port-hopping manner, thus enabling support to multi-party applications as well. The algorithm does not rely on the server having a fixed port open in the beginning, neither does it require from the client to get a "first-contact" port from a third party. We also present an adaptive algorithm,* HOPERAA, *for hopping in the presence of clock-drift, as well as the analysis and evaluation of the methods. The solutions are simple, based on each client interacting with the server independently of the other clients, without the need of acknowledgments or time server. Provided that one has an estimation of the time it takes for the adversary to detect that a port is open and launch an attack, the method we propose does not make it possible to the eavesdropping adversary to launch an attack directed to the application's open port(s).*

1 Introduction

A *Denial of Service*(DoS) attack is an attempt by the attacker to prevent the legitimate users of a service from using that service. One of the main methods that the attacker will use is depleting the computational resources, such as bandwidth, disk space, or CPU time. The situation is even worse with *distributed denial of service*(DDoS) attacks, where multiple compromised machines or zombie agents flood messages or requests of a specific service to the corresponding server in order to make the service unavailable [7]. Common methods to protect systems from DoS and DDoS attacks focus on mitigating packet flooding, as that is the most simple and common method adopted by attackers. Such methods rely on upstream routers that *filter or rate-limit* the malicious traffic [2, 11], or on *secure router-overlays* [5, 1, 4, 10]. These solutions are suitable for filtering distinguishable network flooding but can be ineffective when the attacker changes its strategy and strength to comply to the filtering rules and attack an application directly, as these solutions are general and lack access to application-related information.

When considering network-based applications, a particularly weak point in this context is that they commonly provide some *open port(s)* for communication, making themselves targets for DoS attacks. Adversaries that can eavesdrop messages exchanged by the application can identify open ports and launch *directed* attacks to those that remain open for long enough time –as opposed to *blind* attacks that can be launched to arbitrary ports, even by non-eavesdropping adversaries. Moreover, it is important to note that as an application may e.g. involve complex computations, it could be easier to exhaust its computational resources with small volume of messages, especially when many applications execute in one host and the resources allocated to each application become even smaller.

Therefore a natural question is: what can the applications do to prevent or defend themselves from such situations? This question gets even more important considering the evolution of application overlays, peer-to-peer applications and application-layer networking. A human analogy for the problem is to contrast defense against a distinguishable crowd, that can be taken care by army or police forces, versus protection from sets of seemingly uncoordinated legitimate "agents", that intend to attack some unknown "target", such as a person, an enterprise, etc. The latter may certainly want to ensure their own protection.

This problem was also posed earlier in the literature and a simple and useful approach that has been proposed involves *port-hopping*, inspired from the well-known frequency hopping paradigm used in signal transmission protocols [8]. The application parties communicate via ports that change periodically over time, according to a pattern known to the two parties, such as a (pseudo)random sequence with common seed (cf. [3] and our section on related literature). These solutions have been presented for a client-server pair of communicating parties. Port-hopping in multi-party communication is an interesting challenge. A critical issue involved in port-hopping is synchronizing the communication parties. Two main kinds of coordination mechanisms are presented in the previous work, one is acknowledgment-based and the other one depends on synchronized clocks (cf. section on related work). As acknowledgment loss can cause a situation where a port may remain open for a time interval long enough for an eavesdropping attacker to identify it and launch a *directed attack* to it, and to have synchronized clocks may imply need for synchronization server, which could be the weak point in the system, there is interesting middle ground for investigation and for employing the method on common networking systems.

In this paper we investigate such questions opened in the earlier work, namely supporting port-hopping (i) in the presence of *timing uncertainty*, i.e. *clock-rate drifts*, implying that clock values can vary arbitrarily much with time; and (ii) in *multi-party communication*.

In particular, for dealing with hopping in the presence of clock-rate drifts, we propose the Hopping-Period-Align-and-Adjust algorithm, or HOPERAA for brevity, which is an adaptive algorithm, executed by each client when its hopping period length and alignment drift apart from the server's. For enabling multi-party communication with port-hopping, we present the BIGWHEEL algorithm for a server to support hopping with many clients, without the server needing to keep state for each client individually. The basic idea in both algorithms is that each client interacts independently with the server and considers the server's clock as the point of reference. In this way there is no need for group synchronization which would have raised scalability issues. This makes it also possible to use the method in (e.g. multi-party) applications.

The protocol is presented in terms of client-server type of communication but a symmetric protocol can be run between the two parties with exchanged role, for ensuring the properties in duplex-communication scenario, e.g. similar to the way that TCP applies its reliability algorithms in duplex communication. Our solution is general, because all the mechanisms and algorithms are only based on the clients and the server. The options for the adversary to launch a directed attack to the application's ports after eavesdropping is minimal, since the port hopping period is fixed. Potential message loss due to the hopping period deviation caused by the clock-rate drift can be controlled by adjusting a parameter in the HOPERAA algorithm and we explain how this is done at the corresponding sections.

We proceed by describing in more detail the related work. Subsequently, we give a detailed definition of the problem and the system model in section 2. We continue with the description of the HOPERAA and BIGWHEEL algorithms in sections 3 and 4 respectively. In section 5 we give an analysis of the properties of the methods. Following these we present an experimental study of the algorithms in section 6 and we conclude with section 7.

Related Work

The most closely related results are the port-hopping protocols presented in [3]. The ack-based protocol in that paper is focused on the communication only between two parties, modeled as sender and receiver. The receiver sends back an acknowledgment for every message received from the sender, and the sender uses these acknowledgments as the signals to change the destination port numbers of its messages. Since this protocol is ack-based, time synchronization is not necessary. But note that the acknowledgments can be lost in the network, and this may keep the two parties using a certain port for a longer time. If the attacker gets the port number during this time, then a *directed attack* will be launched under which the communication can hardly survive. To cope with that, a solution that reinitializes the protocol is presented in [3]. The latter solution depends on that the clocks have the same rate; it allows for bounded drift in the clock phases (resulting in bounded differences of clock values) but not their rates (which would imply arbitrary differences of clock values). In [3] the authors also present a rigorous model and analysis of the problem of possible DoS to applications (ports) by an adaptive adversary, i.e. one that can eavesdrop, as in our case, too. The analysis, besides the parts that involve the port-hopping protocols proposed in that paper, also includes a part on the effect of the adversary when it launches blind attacks. As that part of the analysis holds regardless of the applications's defense mechanism, it carries over any setting. Hence, we do not elaborate on that part.

Another port-hopping scheme for the client-server mode was proposed in [6]. There, time is divided into discrete time slots. The clients and the server share a pseudo-random function to compute which port should be used in a certain time slot. This scheme bounds the time offset and the message delay by a constant value l, so there is no time synchronization mechanism. Instead, the valid time of the communication port for a time slot is prolonged both backward and forward by $\frac{1}{2}l$. This scheme shows the basic idea of the time-based port hopping, but it only works when no clock drift exists, which limits its adaptability.

There is a client-transparent approach proposed in [9] which is quite similar to port hopping. This approach uses JavaScript to embed authentication code into the TCP/IP layer of the networking stack, so the messages with invalid authentication code would be filtered by the server's firewall. In oder to defend the DoS attacks, the authentication code changes periodically. There is a challenge server in charge of issuing keys, controlling the number of clients connected with the server and synchronizing the clients with the server as well. Since this approach relies on the challenge server, the protection of the challenge server is quite important. The paper mentions that a cryptographic based mechanism can be used to protect the challenge server, but this was not discussed in detail.

2 Problem and System Model Definitions

We consider the problem that the adversary wants to subvert the communication of client-server applications which communicate via communication channels or, for brevity, *p*orts. Each time some port must be open at the server side to receive the messages sent from the legitimate clients. At the server side there are N ports can be used, so the size of the port number space is N. The server and the legitimate clients share pseudo-random function f_ψ to generate the port number which will be used in the communication. We assume that there exists a preceding authentication procedure which makes the server to distinguish the messages from the legitimate clients. And we also assume that every client is honest which means any execution of the client is based on the protocol and clients can not be compromised by the adversary.

We model the attacker as an *adaptive adversary* which can eavesdrop and launch a bounded number of directed attacks. As mentioned in the related work section, the analysis of the problem presented rigorously in [3], include the analysis of the effect of an adaptive adversary when it launches blind attacks. As that analysis holds regardless of the applications's defense mechanism, it carries over any setting. Hence, we do not elaborate on the case of blind attacks. Regarding the adversary considered here, it is assumed that when it knows that some ports are being used by the server, it can attack at most Q of them and in general it can attack at most Q arbitrary ports of the server simultaneously. For the purpose of the analysis, we bound the strength of the adversary by Q. We also assume that when the adversary attacks a certain port of the server then this port cannot receive any message from the clients. Since the adversary can get the number of the port being used from the clients' messages by eavesdropping but it takes some time to get this information and get ready to launch the directed attack to the port, we model this as the *exposure delay* and bound it by E time units.

For the time model, we assume that each communication party has its local clock, and the clock rate of each local clock is constant. We use the server's clock as the standard one; each client's clock drift is defined as the ratio between its own clock rate and the server's clock rate. We use ρ_C to denote the clock drift of client C. We have to emphasize that in this paper every time variable is related to the server's clock unless otherwise stated. If the server's clock value is t, we use $h_c(t)$ to denote the clock value of client C.

Considering that our solution mitigates DoS attacks at the application layer, we assume that the network is always available meaning that there are no attacks depleting the bandwidth of the server's network. However, the network may lose messages. Finally, for the analysis we assume the maximum delivery latency for messages is μ.

3 Protocol for Single Client Case

We first present the protocol for communication between a single client (denoted by C) and a server (denoted by S). In the subsequent section we describe the BIGWHEEL algorithm that enables the multi-party communication case.

We first give an outline of the protocol and then present more details for its phases. Roughly speaking, after the *contact-initiation phase*, the application data from C to S is sent out through ports of S that change with period L time units of S's clock, corresponding to P_c time units in C's clock (initially $P_c = L$). What is achieved in the contact-initiation phase is that (i) C has succeeded in finding the first port to contact S, without the need of having S keep some "well-known" open ports, nor C relying on a third party to get the port information; and (ii) C has got the seed from S for the pseudo-random function to compute the port sequence. Since C's clock has rate drift compared to S's clock, the periods of C and S may start drifting apart after some time. This would result in message loss, due to the fact that C may send messages to some of S's ports that has been closed or has not opened yet —depending on whether C's clock runs slower or faster that S's clock, respectively. To solve this, C executes the HOPERAA algorithm at intervals that are adaptively computed by C itself. Roughly speaking, S and C timestamp with their clock values the contact-initiation messages during the contact-initiation and the Hopping Period Alignment and Adjustment phases and C uses the timestamps of those messages to estimate an interval where their clock drift may lie in. C then decides the next Hopping Period Alignment and Adjustment execution-interval and also how it can adjust its period so as to cover for the clock rate drift and thus avoid sending messages to closed ports. In the following subsections we describe in detail all the parts of the protocol. Before that, let us give some auxiliary definitions:

Definition 1. *The open ports in the server side for receiving the data messages from the client are called* worker ports. *The open ports in the server side for receiving coordination messages from the client are called* guard ports.

3.1 Contact Initiation Phase

To enable C to initiate contact with S without having S listen at "well-known" ports and without relying on a third party, we propose the algorithm described below.

Algorithm 3.1 Algorithm for C in the initiation stage

```
$T_c \leftarrow undef$
$reply \leftarrow$ false
– sending contact-initiation messages:
while $reply =$ false do
    $I \leftarrow select(I_i | i \in \{1, 2, \ldots, k\})$
    for all $p \in I$ do
        $send \langle init, time, p \rangle$
    end for
    $wait(2\mu + L)$
end while

– receiving reply:
$receive \langle reply, \sigma, time, h_1, t_1 \rangle$
if $reply =$ false then
    $reply =$ true
    $T_c = 0$
    start sending data
end if
```

We divide the range of port numbers into k intervals evenly; the server opens k different guard ports at the same time, one guard port per one interval, and changes them every τ time units but still keeps one guard port in one interval. Here we assume that k can divide N. C sends contact-initiation messages to all the ports in an interval which is randomly chosen. When S receives the contact-initiation message, it replies with the seed σ for computing the next worker port and opens a session for C. If S does not receive any messages from C by the next worker port it will close the session. Since the network is unsafe and can lose messages and since open ports can be disabled by the adversary, C may not get the reply from S. If C does not receive a reply from S it will choose another interval and send contact-initiation messages again until it gets the reply from S. In section 5 we show the bound of the expectation of how many trials C would make to get the reply from S. The algorithms for C and S in the initiation stage are shown in Algorithm 3.1 and Algorithm 3.2, respectively.

3.2 Sending the Application Data

In this stage, C will send data messages to the worker ports of S. We assume that C and S share a common pseudo-random function f_ψ and integer σ as the seed, which is used for computing the sequence of worker ports. The open interval of the worker ports is $L + \mu$, where $L > \mu$. The new worker port will be opened μ time units earlier than the closing time of the old one, as shown in Figure 1. When S receives the contact-initiation messages from C, it will send the reply message at the time when the next worker port is opened, and the integer σ has the value for generating the next worker port. When C gets the integer σ from S's reply, it will send the data messages immediately to the port computed from $f_\psi(\sigma)$. C has a timer T_c which will be assigned to 0 when C receives the reply message from S. T_c increases at the same rate as the local clock of C. The destination port number of the data messages will be recomputed when $T_c = iL$, at every $i \in \mathbb{N}^*$. Since there exists delivery latency, some messages that sent to port p_i (the i-th port in the hopping sequence) may arrive when p_i is closed. So we duplicate these messages, with destination port of the duplicates being p_{i+1}. In our model, if there is no time drift then messages that are sent during the interval $[(i-1)L, iL - \mu]$ should arrive at p_i when p_i is open (otherwise we consider them being lost), so the messages sent both to p_i and p_{i+1} are the messages sent in the interval $[iL - \mu, iL]$. C will end the communication by sending termination message and getting it acknowledged.

Algorithm 3.2 Algorithm for S in the initiation stage

```
– receiving contact-initiation message:
$receive \langle init, time, p \rangle$
$t_1 \leftarrow Time_{now}$
if $session_C = undef$ then
    $open\,(session, C)$
    $h_1 \leftarrow time$
end if
wait until next worker port $p_i$ opens
$send \langle reply, \sigma, timestamp, h_1, t_1 \rangle$
```

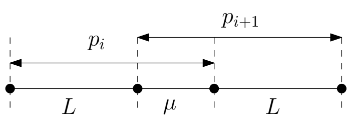

Figure 1. Worker ports' open interval with overlap

Algorithm 3.3 Algorithm for C in data transmission stage

```
$Seq \leftarrow 0$
$P_{old} \leftarrow f_\psi(\sigma)$
$P_{new} \leftarrow f_\psi((\sigma + 1))$
– $S_A$ (*sending the messages*)
while true do
    $send \langle Data, P_{old} \rangle$
    if $(i-1)L \le T_c \le iL - \mu$ then
        $send \langle Data, P_{new} \rangle$
    end if
end while

– $U_A$ (*changing the destination port*)
    $\{T_c = iL\}$
    $P_{old} \leftarrow P_{new}$
    $P_{new} \leftarrow f_\psi((\sigma + i + 1))$
```

3.3 Adaptive Hopping Period

As mentioned in section 2, client C has a constant clock drift ρ related to the server. It may happen that in the application data sending phase, the hopping time of C will drift apart from the server's. This might cause C send messages to a port that is already closed (corresponding to the previous period of S) or is not opened yet (corresponding to the next period of S), depending on whether C's clock is slower or faster that S's. This would imply message loss, so C has to align the hopping time at adaptively chosen time intervals, to control the phenomenon. These are called the HOPERAA *execution-intervals*. In particular, if the client's clock is slower than the server's, which means $\rho < 1$, and if we want to keep the offset of the closing times counted by the server and the client of a worker port within Δ time units, then the HOPERAA execution interval is $\frac{\rho\Delta}{1-\rho}$. If the client's clock is faster than the server which means $\rho > 1$, and we want to keep the offset of the opening times counted by the server and the client of a worker port within Δ time units, then the HOPERAA execution interval is $\frac{\rho\Delta}{\rho-1}$. But the client has no idea about its clock drift related to the server's clock. We suggest a method that exchanges messages (which are piggybacked) with information about the sending and receiving times (timestamped with local clock values) between C and S, to estimate the clock drift. This is illustrated in Figure 4. The procedure of the alignment part in the HOPERAA algorithm is described below:

- The HOPERAA execution interval is initiated to 0. In the contact initiation phase, every contact-initiation message will be attached with the time stamp (the clock value of the client) of its sending time.
- The server will record the sending time $h_c(t_1)$ and the arrival time t_2 of the first contact-initiation message sent by client C.
- When C executes HOPERAA, it will execute the same operations as in the contact initiation phase, the server will add a timestamp of the sending time to every reply message, say t_3, as well as $h_c(t_1)$ and t_2 recorded in the first time that client C initiates contact with the server. The client will record the arrival time of the reply, say $h_c(t_4)$. Then C bounds its clock drift as

$$\frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2 + 2\mu} \le \rho \le \frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2} \qquad (1)$$

- If both the lower bound and the upper bound are smaller than 1, then the new HOPERAA execution interval is $\frac{\rho\Delta}{1-\rho}$, where ρ will be replaced with the lower bound. If both bounds are greater than 1, then the new HOPERAA execution interval is $\frac{\rho\Delta}{\rho-1}$, where ρ will be replaced with the upper bound. If the lower bound is smaller than 1 but the upper bound is greater than 1, then the new HOPERAA execution interval will be

$$\min\left\{\frac{\rho_l\Delta}{1-\rho_l}, \frac{\rho_u\Delta}{\rho_u - 1}\right\},$$

  where ρ_l and ρ_u denote the lower and upper bound respectively.

In the next section we will show the correctness of formula 1, and we will also show that every time C estimates its clock drift related to the server, it will get a better bound than the one it got from the previous estimation. The adjustment part of HOPERAA is described in the following items:

- If $1 \le \rho_l \le \rho_u$, then adjust P_c to $L \cdot \rho_l$, and the HOPERAA execution interval is set to $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l}$.
- If $\rho_l \le \rho_u \le 1$, then adjust P_c to $L \cdot \rho_u$, and the HOPERAA execution interval is set to $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l}$.
- Otherwise, do nothing.

Before C adjusts P_c, it has to know whether its clock rate is faster or slower than the server, otherwise it has no idea whether to shorten P_c or extend P_c. Since the bounds of the drift improve monotonically, (cf. section 5) gradually C will know whether to shorten P_c or extend P_c. Intuitively, if the clock drift of C is big then it takes few rounds of drift estimation to let C make the adjustment to P_c, since the influence of the message delivery latency is relatively small. If the clock drift is very close to 1 then it may take more rounds to let C make the decision. Consider an extreme example that the clock drift is equal to 1 meaning that the client's clock rate is equal to the server, then the client can never know whether its clock rate is faster or slower than the server. But on the other hand, if ρ is close to 1, it means that $\frac{\rho\Delta}{1-\rho}$ or $\frac{\rho\Delta}{\rho-1}$ can not be very short and C does not have to do the alignment frequently before the adjustment of P_c.

4 Supporting Multiple Clients per Server

The extension to multiple clients per server is based on a simple idea: Each client follows the server's hopping procedure; since each client considers the server's clock as reference clock, it can interact with the server independently of the other clients, and send the application's data to S's port which is active each time. For scalability reasons it is desirable that the server has more than one worker ports open each time (but still a small constant number of those), so as to balance the load among them. Moreover, by having the same hopping period but different phases in the corresponding hopping sequences, such a method can manage to bound better the time it takes for each client to initiate contact with the server.

Intuitively, and as the name also suggests, the BIG-WHEEL algorithm proposed here, aiming at meeting the aforementioned goals, functions as the Big Wheel rides at amusement parks: clients queue for the next available compartment —each compartment represents a hopping sequence; compartments are deployed in a way that aims at balancing the load among them and also at minimizing the clients' waiting times to initiate contact with the server. The procedure is described in more detail below.

Algorithm 4.1 Algorithm for S using several sequences.

```
Buffer B stores valid contact-initiation messages.
whenever$\left(Time_{now} = OpenTime_{p_j^i}\right)$
$\lambda \leftarrow$ the corresponding value for sequence $j$
$\sigma \leftarrow$ the corresponding value for $p_j^i$
for all the clients of messages in B do
    if $session_C = undef$ then
        $open\,(session, C)$
        $h_1 \leftarrow$ timestamp of the contact-initiation message
    end if
    $send\,\langle reply, \sigma, timestamp, h_1, t_1, \lambda\rangle$
end for
Clear B
```

Let the clock drift of client C be denoted by ρ_C and consider a trivial big wheel with one compartment. In such a setting the server opens a session for each client whose contact-initiation messages are received, and closes the session after the corresponding termination message's arrival. Again in the same setting, since every client uses the same pseudo-random function to generate the destination port number, one worker port has to receive the messages from all the active clients. Moreover, when the server receives a contact-initiation message, it will not send its reply (so that the client starts its periods in-sync with S) until the next worker port is open, which increases the clients waiting time by at most L time units. In order to afford more clients and also decrease the maximum waiting time for a client, we can add one more parameter into the pseudo-random function, say λ: f_ψ will generate different port number sequences if different values of λ are given. Suppose there are $m \geq 2$ values for λ (i.e. we have m compartments in the big wheel). It means that S supports m port number sequences. Let us use p_j^i to denote the i^{th} worker port in the j^{th} port number sequence, where $0 \leq j \leq m-1$. The server will change the worker ports according to each sequence in the following way: If the open time of port p_0^i is t_i then the open time of port p_j^i is $t_i + \frac{jL}{m}$. The open interval of every worker port is still $L + \mu$. Figure 2 shows the situation of $m = 3$ and the open time of p_0^i is t.

Based on this mechanism, when the server receives a contact-initiation message from a client, it will send the reply at the closest open time of a worker port, including λ with the corresponding value for the sequence to which that worker port belong. A pseudocode is shown in Algorithm 4.1. By using multiple port number sequences, the maximum waiting time for a client in the contact initiation phase can be decreased to $2\mu + \frac{L}{m}$ time units.

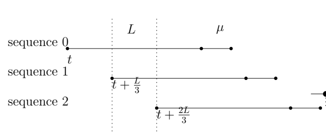

Figure 2. The open intervals of port p_0^i, p_1^i and p_2^i.

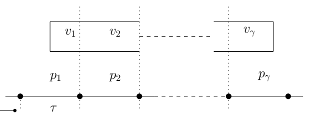

Figure 3. Arrival duration covers γ changing period of guard ports.

5 Analysis of the Protocol

We start by some auxiliary definitions that are useful in this section:

- We say a client gets a *successful access* to the server, when at least one of its contact-initiation messages are received by the server.
- A *contact-initiation trial* is a part of an execution starting with a client randomly choosing an interval of the port space and send contact-initiation messages to each of the ports in that interval and ending with the reply from the server or reaching a time out of waiting.
- We say that the adversary launch a *blind attack* if the adversary arbitrarily chooses and attacks Q ports of the server simultaneously.

First, we analyze the contact initiation part of the protocol. Since without initiation the client cannot hop together with the server and since the guard ports cannot be fixed, we have to hop guard ports but in a range which is smaller than N. Note that if we fix this range then the adversary can learn it from the contact-initiation messages of the client (because client always send contact-initiation messages to that range) and then launch a directed attack to the application's open port(s). In our protocol, we divide the port number space into k intervals, $I_i = \left\{p_j | i\frac{N}{k} \leq j \leq (i+1)\frac{N}{k} - 1\right\}$, $i = 0, 1, 2, \ldots, k-1$. Since a client has no idea which port is open as the guard port in I_i, it sends contact-initiation messages to every port in the interval it chooses and expects that the server can receive one of them. In the presence of delivery latency and guard ports' changing, the contact-initiation message sent to the current guard port of the chosen interval may miss the port, then this *contact-initiation trial* may fail. We show how to bound this probability and also give the corresponding experiment result in section 6.

Lemma 1. *If the adversary launches a blind attack, the probability that it disables the guard port in the interval*

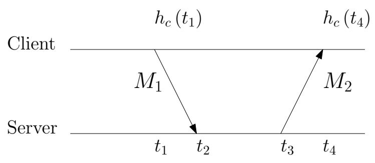

Figure 4. Messages exchange and associated times and timestamps.

chosen by the client in the synchronization stage is $\frac{Q}{N}$, even if the adversary knows the partition of the ports space.

Proof. Suppose the adversary knows the partition, and it disables q_i ports in interval I_i, $i = 0, 1, 2, \ldots, k-1$. Since we have k intervals, and every interval has N/k ports, the probability that the adversary disables the guard port in the interval chosen by the client, say I', is

$$\sum_{i=0}^{k-1} Pr[I_i = I'] = \sum_{i=0}^{k-1} \frac{1}{k} \cdot \frac{q_i}{\frac{N}{k}} = \frac{\sum_{i=0}^{k-1} q_i}{N}$$

Since $\sum_{i=0}^{k-1} q_i = Q$, the probability is $\frac{Q}{N}$. If the adversary does not know the partition, then it will attack arbitrary ports, so the probability that the guard port is under attack is $\frac{Q}{N}$. □

Based on lemma 1 we will give a lower bound on the probability that one contact-initiation trial can lead to successful access.

Lemma 2. *Suppose the adversary launches a blind attack, and the size of the port space is N, and there is no message loss during the transmission but there exists delivery latency, then the probability that one contact-initiation trial can lead to successful access is at least $1 - \left(\frac{1}{e}\right)^F$, where $F = \frac{N-Q}{N}$ is the fraction of non disabled ports .*

Proof. (Sketch) Set V be the number of ports in one interval. In one contact-initiation trial, the client will send V messages to the interval it chooses, one message to one port in that interval. As shown in Figure 3, the arrival duration of those V messages may cover γ changing periods of the guard ports. We use p_i, $1 \le i \le \gamma$ to denote the guard port for period i, and use v_i to denote the number of messages that arrive within period i. We assume that each $v_i > 0$, since if $v_i = 0$, the probability that p_i receives the message sent to it will be definitely zero. The probability that the server does not receive any contact-initiation message is

$$Pr_{[\text{trial fails}]} = \Pi_{i=1}^{\gamma} Pr_{[p_i \text{ does not receive the message sent to it}]},$$

$$= \Pi_{i=1}^{\gamma} \left(Pr_d + Pr_l - Pr_d \cdot Pr_l\right),$$

where Pr_d is the probability that p_i is disabled by the adversary, Pr_l is the probability that the message sent to p_i does not arrive within period i. So we have

$$\begin{aligned} Pr_{[\text{trial fails}]} &= \Pi_{i=1}^{\gamma} \left(\frac{Q}{N} + \frac{V - v_i}{V} - \frac{Q}{N} \cdot \frac{V - v_i}{V}\right) \\ &= \Pi_{i=1}^{\gamma} \left(1 - \frac{(N-Q)\, v_i}{N \cdot V}\right). \end{aligned}$$

Using the method Lagrange multipliers, we know that $Pr_{[\text{trial fails}]}$ has the maximum value when $v_1 = v_2 = \ldots = v_\gamma = \frac{V}{\gamma}$. So we have

$$Pr_{[\text{trial fails}]} \le \left(1 - \frac{N-Q}{N \cdot \gamma}\right)^{\gamma} = \left(1 - \frac{1}{\frac{N \cdot \gamma}{N-Q}}\right)^{\frac{N-Q}{N}\left(\frac{N \cdot \gamma}{N-Q}\right)}.$$

Since we know that function $\left(1 - \frac{1}{x}\right)^x$ is a monotonically increasing but bounded function when $x > 0$, and the limit is $\frac{1}{e}$ when $x \to +\infty$, where e is the mathematical constant and $e \approx 2.72$. Hence, we have

$$Pr_{[\text{trial fails}]} \le \left(\frac{1}{e}\right)^F, F = \frac{N-Q}{N}$$

then it is obvious to see that the the probability that one contact-initiation trial can lead to successful access is at least $1 - \left(\frac{1}{e}\right)^F$.

□

Corollary 1. *The expected value of the number of contact-initiation trials is at most $\frac{1}{1-\left(\frac{1}{e}\right)^F}$. The expected value of the number of contact messages used in the contact-initiation phase by one client is at most $\frac{N}{k} \cdot \frac{1}{1-\left(\frac{1}{e}\right)^F}$, where k is the number of intervals in the port space.*

Recall the assumption that the adversary can do eavesdropping and launch directed attacks to the open ports, but this takes it E time units from the time it gets a data message from the client. Our protocol aims at keeping the open interval of the worker ports smaller than E. But remember that some client's clock may be faster than the server, and can send messages to a worker port before its opening which will let the adversary get the port number earlier than supposed. But the clients will execute HOPERAA algorithm to align the hopping period to keep itself not drift apart from the server more than Δ time units. So we get the following:

Lemma 3. *If $E > L + \mu + \Delta$, then the adversary can not launch a directed attack to an open port of our protocol.*

The next lemma shows the correctness of formula 1 used in HOPERAA to estimate the clock drift.

Lemma 4. *Suppose we use server's clock as standard, and consider that the client sent message M_1 at time t_1 with the timestamp $h_c(t_1)$, received by the server at time t_2. Consider also that the server sent later one message M_2 at time*

t_3 with timestamp t_3, which is received by the client at time t_4 which is $h_c(t_4)$ according to the client's clock. Then we have

$$\frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2 + 2\mu} \leq \rho \leq \frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2},$$

where ρ is the client's clock drift related to the server's clock, and μ the maximum of the message delivery latency.

Proof. (Sketch) Consider Figure 4. According to the clock drift definition, we have

$$\rho = (h_c(t_4) - h_c(t_1)) / (t_4 - t_1) = \frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2 + d_1 + d_2},$$

where d_1, d_2 are the delivery latencies of M_1 and M_2 respectively. Since $0 \leq d_1+d_2 \leq 2\mu$, the lemma follows. □

From Lemma 4, we can see that the influence of the message delays on the clock drift estimation will decrease when the value of t_3-t_2 is increased, i.e. as the execution evolves and C has repeated the HoPerAA algorithm several times. In our protocol, the server keeps $h_c(t_1)$ and t_2 unchanged, so $t_3 - t_2$ is equal to the time elapsed from the first initiation. Hence the value of $t_3 - t_2$ used in every HoPerAA execution will be greater than the value used in the previous time. Hence, the client will be converging to better and better lower and upper bounds of its clock drift as the execution progresses.

Lemma 5. *Using the* HoPerAA *algorithm, consider the client starts sending data messages to port p at time t (according to the server's clock) and changes the destination port at time t' (according to the server's clock). Then t will not be Δ time units smaller than the corresponding opening time of port p by the server, and t' will not be Δ time units greater than the corresponding closing time of port p by the server.*

Proof. (Sketch) Suppose the client's clock drift is ρ, then the client uses $\frac{1}{\rho}$ time unit to count 1 time unit, the offset will be $\left|1 - \frac{1}{\rho}\right|$. So if we want to keep client's hopping times not drifting Δ time units away from the server's, then the HoPerAA execution interval should be $\frac{\rho\Delta}{|1-\rho|}$. Since $\rho_l \leq \rho \leq \rho_u$, $\frac{\rho\Delta}{|1-\rho|} \geq \min\left\{\frac{\rho_l\Delta}{|1-\rho_l|}, \frac{\rho_u\Delta}{|1-\rho_u|}\right\}$, and we use $\min\left\{\frac{\rho_l\Delta}{|1-\rho_l|}, \frac{\rho_u\Delta}{|1-\rho_u|}\right\}$ as the HoPerAA execution interval, the client hopping times will not drift away from the server's Δ time units.
Suppose that the client knows that its clock drift is bigger than 1, then it will change the hopping period to $L \cdot \rho_l$, which is $\frac{\rho_l}{\rho}L$ time units according to the server's clock then the difference is $\left(1 - \frac{\rho_l}{\rho}\right)L$, so the HoPerAA execution interval should be $\frac{\rho_l\Delta}{1-\frac{\rho_l}{\rho}}$, and the client uses $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l}$ as the HoPerAA execution interval. Since $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l} \leq \frac{\rho_l\Delta}{1-\frac{\rho_l}{\rho}}$, the difference of the hopping times of the client will not drift Δ time units away from the server's. The situation is symmetric when the client knows its clock drift smaller than 1, hence the lemma follows. □

Overhead induced by the HoPerAA **algorithm**: When a client executes the HoPerAA algorithm, it performs the same operations as in the contact initiation phase. Hence, the expected message overhead of HoPerAA is also $\frac{N}{k} \cdot \frac{1}{1-\left(\frac{1}{e}\right)^F}$ for every time that it is executed. In section 6 we will see that the HoPerAA execution interval becomes significantly longer as the execution evolves. Hence, the overall overhead becomes smaller in the course of the execution.

Next we focus on the analysis of BigWheel. In the analysis, we give bounds of the expectation of the number of worker ports being open at the same time, and shows the the probability that at least one of them is under attack when the adversary launches a blind attack.

Lemma 6. *If we have m port sequences in our system, and let M be the number of worker ports being open at the same time, then the expectation of M can be bounded by the following formula:*

$$\sum_{n=1}^{m} n \cdot \frac{\binom{N}{n} S_n(m)}{N^m} \leq E(M) \leq \sum_{n=1}^{2m} n \cdot \frac{\binom{N}{n} S_n(2m)}{N^{2m}}, \qquad where$$

$$S_n(x) = \begin{cases} 1, & n = 1, \\ n^x - \sum_{i=1}^{n-1} \binom{n}{i} S_i(x), & n \geq 2, x \in \{m, 2m\} \end{cases}$$

Proof. (Sketch) Since we assume that every port sequence is randomly generated, it is possible that sometimes several sequences choose the same port as the worker port. The expectation of the number of worker ports opening at a specific time point t is:

$$M = \sum n \cdot Pr\left[n \text{ worker ports are open at time } t\right],$$

where $1 \leq n \leq \max\{\text{all possible values of } M\}$. The size of the port space is N and we have m port sequences. Each sequence chooses a port number randomly. If we use the choices of the sequences to form a string (e.g. $p_0p_1p_2 \cdots p_{m-1}$), then the total number of different strings is N^m. Note that the probability of n worker ports open at time t is the ratio between the number of different strings such that exactly n ports are chosen by m sequences and the total number of different strings. The number of combinations of choosing n ports from N ports is $\binom{N}{n}$. The recursive function $S_n(x)$ is used for computing the number of different strings such that all of the n ports are chosen by x sequences in a way that each of the sequence chooses one port from the n ports. Remember that the open intervals of the old worker port and the new worker port in a sequence have an overlap, hence that can be regarded as the server opens worker ports for $2m$ sequences. Hence, we use $S_n(m)$ and $S_n(2m)$ for the lower and upper bound respectively. Now the only thing left is proving the correctness of $S_n(x)$. The proof of $S_n(x)$ is very simple: Note that for specific n ports and x sequences, where $x \geq n$, each sequence chooses a port from these n ports. The number of different strings such that all the n ports are chosen by the x sequences equals to the difference between the total number of different strings and the number of strings such that not all the n ports are chosen by the x sequences. So this difference can be expressed by

$$n^x - \sum_{i=1}^{n-1} \binom{n}{i} S_i(x).$$

Note that when $n = 1$, the number of different strings such that all of the sequences choose this port is trivially one, hence the lemma follows. □

Lemma 7. *Suppose the adversary launches blind attack, and it can disable Q ports simultaneously, if M worker ports are open at the same time, then the probability that at least one of the worker ports is under attack is*

$$1 - \binom{N-M}{Q} / \binom{N}{Q},$$

where N is the size of the port number space.

Proof. (Sketch) Since the probability that none of the worker ports are under attack is $\binom{N-M}{Q}/\binom{N}{Q}$, the lemma follows. □

6 Experimental Study

In this section we present results from several simulations experiments which can illustrate further some important properties of our protocol. The first experiment simulates the contact-initiation phase. We choose $N = 65536$, and $k = 64$, which translates to 64 intervals in the port space and each one having 1024 ports that can be used. We vary the strength of the adversary from $Q = 10000$ to $Q = 50000$, and for each value of Q we let the client perform 50 repetitions of the contact-initiation phase, and then record the number of trials of each contact-initiation phase. We computed the average number of trials that a client has to perform over all these contact-initiation phases. Figure 5 shows both the experimental outcome and the upper bound of the expectation computed in corollary 1. The average number of trials grows with Q, but we can see that even for $Q = 50000$, the average number of trials during the contact-initiation phase is still not high (4.2 trials).

In the second experiment we study how the HoPerAA execution interval grows (i.e. the protocol overhead decreases) as a function of the number of executions of the HoPerAA algorithm under different values of clock-rate drift. As shown in Figure 6, we choose $\Delta = 0.1L$ and the HoPerAA execution interval is shown in units of the message latency bound, μ. In most cases, the client will know whether its clock rate is faster or slower than the server's after executing HoPerAA 3 times, then it will adjust its hopping period. From Figure 6, it is easy to see that the HoPerAA execution interval grows exponentially with the number of HoPerAA executions.

The last experiment considers the effect of drifts and studies the percentage of messages received by the server under the settings where $Q = 0$ (i.e. the receiving percentage is only affected by HoPerAA), $\rho \in \{0.6, 0.7, 0.8, 0.9, 1.1, 1.3, 1.5\}$ and $\Delta \in \{0.1L, 0.3L\}$ and $\mu = 100$ ms. For each value combination of ρ and Δ, we let the client execute 10 times the HoPerAA algorithm and we record the percentage of the messages received by the server. The results of this experiment are shown in Figure 7. We can see that the percentage of messages received is very high (above 95%) for all time drifts used when $\Delta = 0.1L$. When we choose $\Delta = 0.3L$, the percentage of messages received is close to 90% for $\rho > 1$ and around 95% for $\rho < 1$. The reason for getting better performance when $\rho < 1$ is because the client uses the $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ as the HoPerAA execution interval. From the proof of lemma 5 we can see that this value is smaller than the expected one which is $\frac{\rho \rho_l \Delta}{\rho - \rho_l}$ for $\rho > 1$ and $\frac{\rho \rho_u \Delta}{\rho_u - \rho}$ for $\rho < 1$. This means that the client will pause the message sending process (in order to execute the HoPerAA algorithm) before the hopping time offset reaches Δ time units. And in particular, when $\rho < 1$ this phenomenon is even prominent since the message delivery latency bound μ is much bigger than the actual message delivery latency (i.e. bounded by approximately 50 ms in these experiments—recall that μ was set to 100 ms), and ρ_u is closer to ρ than ρ_l. Hence, the ratio between the HoPerAA execution interval $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ and $\frac{\rho \rho_u \Delta}{\rho_u - \rho}$ is smaller than that between $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ and $\frac{\rho \rho_l \Delta}{\rho - \rho_l}$, which causes the receiving percentage for $\rho < 1$ to be higher than the respective one for $\rho > 1$.

7 Conclusions

In this work we investigate the application-level protection against DoS attacks. More specifically, supporting port-hopping in the presence of timing uncertainty and enabling multi-party communications. In particular, we presented an adaptive algorithm for dealing with port hopping in the presence of clock-rate drifts (such a drift implies that the peer's clock values may differ arbitrarily with time). For enabling multi-party communications with port-hopping, we present an algorithm for a server that supports port hopping with many clients, without the server needing

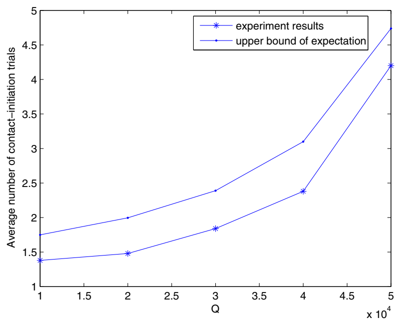

Figure 5. The average number of contact-initiation trials in one contact initiation phase, $\tau = 1000$ millisecond,and client sending speed is 1 message per millisecond

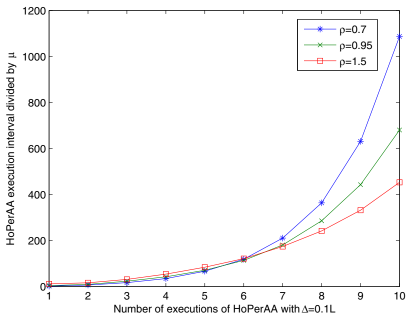

Figure 6. The HOPERAA execution interval grows with the number of HOPERAA executions. The interval is expressed in terms of μ.

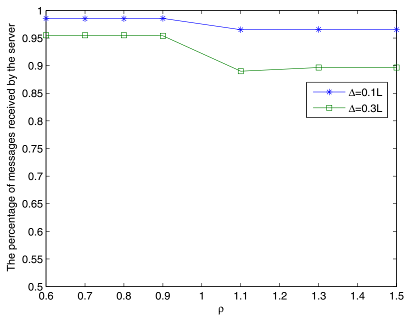

Figure 7. The receiving message percentage affected by the drift.

to keep the state for each client individually. This makes it also possible to use the protocol in multi-party applications. The methods do not induce any need for group synchronization which would have raised scalability issues. The options for the adversary to launch a directed attack to the application's ports after eavesdropping is minimal, since the port hopping period of the protocol is fixed. We have presented an analytical and experimental evaluation of the algorithmic components of the protocol, the latter using simulation studies.

References

[1] D. G. Andersen. Mayday: distributed filtering for internet services. *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 3–3, 2003.

[2] K. Argyraki and D. R. Cheriton. Active internet traffic filtering: real-time response to denial-of-service attacks. *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, 2005.

[3] G. Badishi, A. Herzberg, and I. Keidar. Keeping denial-of-service attackers in the dark. *IEEE Trans. Dependable Secur. Comput.*, 4(3):191–204, 2007.

[4] X. Fu and J. Crowcroft. Gone: an infrastructure overlay for resilient, dos-limiting networking. *ACM NOSSDAV '06: Network and Operating Systems Support for Digital Audio and Video*, 2006.

[5] A. D. Keromytis, V. Misra, and D. Rubenstein. Sos: secure overlay services. *SIGCOMM Comput. Commun. Rev.*, 32(4):61–72, 2002.

[6] H. Lee and V. Thing. Port hopping for resilient networks. *Vehicular Technology Conference, 2004. VTC2004-Fall. 2004 IEEE 60th*, 5:3291–3295, 2004.

[7] J. Mirkovic and P. Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[8] S. S. Scene. http://sss-mag.com/ss.html 2008-03-01.

[9] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu. A client-transparent approach to defend against denial of service attacks. *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 61–70, 2006.

[10] A. Stavrou and A. D. Keromytis. Countering dos attacks with stateless multipath overlays. *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 249–259, 2005.

[11] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. *IEEE Security and Privacy Symposium*, page 130, 2004.