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

Zhang Fu, Marina Papatriantafilou, and Philippas Tsigas

Abstract—Network-based applications commonly open some known communication port(s), making themselves easy targets for (distributed) Denial of Service (DoS) attacks. Earlier solutions for this problem are based on port-hopping between pairs of processes which are synchronous or exchange acknowledgments. However, acknowledgments, if lost, can cause a port to be open for longer time and thus be vulnerable, while time servers can become targets to DoS attack themselves. Here, we extend port-hopping to support multiparty applications, by proposing the BIGWHEEL algorithm, for each application server to communicate with multiple clients in a port-hopping manner without the need for group synchronization. Furthermore, we present an adaptive algorithm, HOPERAA, for enabling hopping in the presence of bounded asynchrony, namely, when the communicating parties have clocks with clock drifts. 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(s). Further, they do not rely on the application having a fixed port open in the beginning, neither do they require the clients to get a "first-contact" port from a third party. We show analytically the properties of the algorithms and also study experimentally their success rates, confirm the relation with the analytical bounds.

Index Terms—Clock drift, data communication, denial of service attack, reliability, application.

1 Introduction

Internet grows rapidly since it was created. Via the Internet infrastructure, hosts can not only share their information, but also complete tasks cooperatively by contributing their computing resources. Moreover, an end host can easily join the network and communicate with any other host by exchanging packets. These are encouraging features of the Internet, *openness* and *scalability*. However, attackers can also take these advantages to prevent legitimate users of a service from using that service by flooding messages to the corresponding server, which forms a *Denial of Service* (DoS) attack.

There are several types of such attacks. An attacker can possibly launch a DoS attack by studying the flaws of network protocols or applications and then sending malformed packets which might cause the corresponding protocols or applications getting into a faulty state. An example of such attacks is *Teardrop* attack [2], which is sending incorrect IP fragments to the target. The target machine may crash if it does not implement TCP/IP fragmentation reassembly code properly. This kind of attacks can be prevented by fixing the corresponding bugs in the protocols or applications. However, the attacker does not always have to do its best to study the service if it wants to make it unavailable. It can just flood packets to keep the server busy with processing packets or cause congestion in the victim's network, so that the server might not have the ability to handle the packets from legitimate hosts or even cannot receive packets from them. In order to deplete the victim's key resources (such as bandwidth and CPU time), the attacker has to aggregate a big volume of malicious traffic. Most of the time, the attacker collects many (could be millions) of *zombie machines* or *bots* to flood packets simultaneously, which forms a *Distributed Denial of Service*(DDoS) attack.

Most of the methods that protect systems from DoS and DDoS attacks focus on mitigating malicious bandwidth consumption caused by packets flooding, as that is the most simple and common method adopted by attackers. Those methods may mitigate DDoS attacks reactively by identifying the malicious traffic and informing the upstream routers to *filter or rate-limit* the corresponding traffic [3], [4], [5], [6], [7], [8]; they may also mitigate DDoS attacks by deploying *secure overlays* [9], [10], [11], [12], or by distinguishing the legitimate traffic with valid *network capabilities* [13], [14], [15], [16].

These solutions are suitable for filtering bandwidth attacks. However, the attacker may change its strategy and attack an application directly, especially when the application involves complex computations. It could be easier to exhaust its computational resources with small volume of messages. Therefore, the malicious traffic against an application has usually small volume and it is difficult to be detected [17]. Defense methods mentioned above may help, but they might not be efficient and accurate with respect to a certain application, as they lack application-related information. Considering there are numerous applications, it would be very expensive and impractical for traffic monitors to keep information for every application.

• *The authors are with the Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. E-mail: {zhafu, ptrianta, tsigas}@chalmers.se.*

Manuscript received 23 Feb. 2011; revised 21 Dec. 2011; accepted 17 Jan. 2012; published online 26 Jan. 2012.
Recommended for acceptance by X. Wang.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2011-02-0031.
Digital Object Identifier no. 10.1109/TDSC.2012.18.

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. Therefore, one question is worthy of being investigated: How can network-based applications defend DDoS attacks by themselves? This question gets even more important considering the evolution of application overlays, peer-to-peer applications and application-layer networking.

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 have the ability of eavesdropping messages exchanged by the application can identify open ports and launch *directed* attacks to those ports—as opposed to *blind attacks* that can be launched to arbitrary ports, even by noneavesdropping adversaries. This problem was also posed earlier in the literature and a simple and useful approach was proposed, namely, *port-hopping*: the application parties communicate via ports that change periodically over time, according to a pattern known by both the sender and receiver, such as a (pseudo)random sequence with common seed (cf. [18] and our section on related literature). This method was inspired from the well-known frequency hopping paradigm used in signal communication protocols [19]. The focus in that area is to find the hopping sequences with the optimal Hamming Correlation Properties [20], [21], [22].

One of the critical issues involved in port-hopping is synchronizing communication parties. Two main kinds of synchronization mechanisms are presented in the previous work, one is acknowledgment-based and the other one depends on synchronized clocks (cf. section on related work). Acknowledgment loss can cause a situation where a port may remain open for a time interval long enough for an eavesdropping attacker to identify and launch a *directed attack* to it. Having synchronized clocks may imply need for synchronization server, which could be the weak point in the system. Hence, these imply challenges for investigation to deploy the method in common networking systems, especially when multiple communication parties are involved.

With the synchronization issue in mind, our goals in this work are to support port-hopping 1) in the presence of *timing uncertainty*, i.e., *clock-rate drifts*, implying that clock values can vary arbitrarily much with time; and 2) in *multiparty communication*. In order to deal 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 to adjust its hopping period length and align its hopping time with the server. To enable multiparty communication with port-hopping, we propose 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; moreover, the server does not need to maintain a state for each client, since the main responsibility for the coordination is assigned to the client(s). As in, e.g., TCP, a client of one session can be a server for another (possibly concurrent) session, hence the solution proposed here fits for symmetric use, though the protocol is presented in client-server type. According to the properties of our algorithm, there is no need for group synchronization which would raise scalability issues.

Our solution is general, because the mechanisms and algorithms are only based on the clients and server(s). It can be a complementary mechanism to the ones against bandwidth attacks. By adjusting the hopping period (i.e., roughly the time that communication ports remain open), the situation that the adversary is able to launch a directed attack to the application's ports after eavesdropping is limited. Potential message loss due to the hopping period deviation caused by the clock-rate drifts can be controlled by adjusting a parameter in the HOPERAA algorithm. The message overhead for setting connections between communication parties is bounded and its average overhead is observed to follow an exponential style of decay.

The paper is organized as follows: in Section 2, we give a detailed definition of the problem and the system model. We continue with the description of the HOPERAA and BIGWHEEL algorithms in Sections 3 and 4, respectively. In Section 5, we give an analysis on properties of the methods. We validate some of our analysis and give complementary experimental study in Section 6. In Section 7, we discuss some implementation issues. Finally, we conclude in Section 8.

1.1 Related Work

There are many *network-based* solutions against DDoS attacks. These solutions usually use routers or overlay networks to filter malicious traffic. A good survey about network-based defense mechanisms against DDoS attacks is presented by Peng et al. [23]. In this paper, we focus on *application-based* mitigation.

Badishi et al. [18] propose an ack-based port-hopping protocol focusing 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 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 longer time. If the attacker gets the port number during this time, then a *directed attack* can be launched under which the communication can hardly survive. Hari and Dohi present an analysis on the sensitivity of this protocol to attacks [24]. To cope with that, Badishi et al. [18] also propose a solution that reinitializes the protocol. With reinitializing periodically, the sender and receiver can use new seeds of the pseudorandom function to generate different port number sequences, so that the port number sequence used for communication is changed periodically. Thus, even though the attacker can launch the directed attack due to the lost of acknowledgment packets, the sender and receiver can continue the communication by reinitializing the protocol. This reinitialization is based on

an assumption that the difference of clock values of the two communication parties is bounded in order to make the sender and receiver reinitialize around the same time. In this work, we assume that the differences of clock values can be arbitrary, but the clock rate of each communication party is constant.

In [18], Badishi et al. 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 this paper, too. The analysis, besides the parts that involve the port-hopping protocols proposed in that paper, also includes a part analyzing the effect of the adversary's different strategies for launching blind attacks. The goal of the attacker is to decrease the probability that a client's message is received by the server (it is called delivery probability in the paper) as much as possible. The authors showed a lower bound that the attacker cannot decrease the delivery probability below that. This lower bound is based on the capacity of a port for receiving messages and also the adversary's ability to flood messages. As those results hold regardless of the applications's defense mechanism, they carry over any setting, including the methods proposed in this paper.

Lee and Thing [25] propose another port-hopping scheme for the client-server mode. In their mechanism, time is divided into discrete time slots. The clients and the server share a pseudorandom function to compute which port should be used in a certain time slot. The authors assume that the time offset plus the message delay is bounded by a constant value l, so there is no time synchronization mechanism needed. Instead, the valid open 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 still it is based on synchronized clock values.

Similar as port-hopping, Srivatsa et al. [26] propose a client-transparent approach. This approach uses JavaScript to embed authentication code into the TCP/IP layer of the networking stack, so the messages with invalid authentication code will be filtered by the server's firewall. In oder to defend the DoS attacks, the authentication code changes periodically. A challenge server is deployed whose responsibility is 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, although this was not discussed in detail. In our work, we do not use any third party for time synchronization.

2 Problem and System Model Definitions

We focus on the problem that an adversary wants to subvert the communication of client-server application by attacking their communication channels or, for brevity, *ports*. At each time point, some port must be open at the server side to receive the messages sent from legitimate clients. At the server side, the size of port number space is N, meaning that there are N ports that the server can use for communication. The server and the legitimate clients share a pseudorandom function f_ψ to generate the port numbers which will be used in the communication. We assume that there exists a preceding authentication procedure which enables the server to distinguish the messages from the legitimate clients. We also assume that every client is honest which means any execution of the client is based on the protocol and clients will not reveal the random function to the adversary.

The attacker is modeled as an *adaptive adversary* which can eavesdrop and attack a bounded number of ports simultaneously. For the purpose of the analysis, we bound the strength of the adversary by Q, meaning that it can attack arbitrarily at most Q ports of the server simultaneously. We also assume that when the adversary attacks a certain port of the server then this port cannot receive any message from the clients. As mentioned in the related work section, Badishi et al. [18] presented an analysis about the effect of adversary's different strategies when it launches blind attacks that disable open ports only partially. We do not elaborate on this again.

The adversary can get the number of the port being used from the clients' messages by eavesdropping, however 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. So by changing the communications ports, one can limit the adversary's ability to launch directed attacks effectively (see Lemma 3).

Unlike previous work's assumptions about time synchronization, 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.

Since our solution mitigates DoS attacks at the application layer, it cannot defeat the bandwidth-based attacks. So following the assumptions of previous work [18], [26], we also assume that the network is always available meaning that there are no bandwidth-based attacks. However, the network may lose messages. Finally, for the analysis, we assume the maximum delivery latency for messages is μ. This can be a configurable parameter of the protocol, depending on the deployment (see Section 7 for a discussion of this issue).

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 multiparty communication. Without loss of generality, one server is considered throughout the presentation for readability issues. For the situation of multiclient and multiserver, clients and servers follow the algorithms for the clients and servers, respectively.

3.1 Overview

Roughly speaking, the whole port hopping mechanism consists of three parts: the *contact-initiation part*, the *data transmission part*, and the *resynchronization/adjustment part* which is controlled by the Hopping Period Alignment and Adjustment (HOPERAA) algorithm.

What is achieved basically by the client C in the contact-initiation part is 1) C has succeeded in finding the first port to contact S, without the need of having S keep some "well-known" ports open, nor C relying on a third party to get the port information; and 2) C gets the seed from S for the pseudorandom function to compute the port sequence. After the contact-initiation part, the application data from C to S is sent out to the open ports of S that change every L time units of S's clock, corresponding to L_c time units in C's clock (initially $L_c = L$).

Since C's clock has a drift related to S's clock, if they have different clock rates, then the lengths of their hopping periods will deviate from each other. This would result in message loss, due to the fact that C may send messages to some of S's ports which have been closed or have not been opened yet—depending on whether C's clock runs slower or faster than S's clock, respectively. To solve this, C executes the HOPERAA algorithm to adjust its hopping period. Roughly speaking, S and C attach timestamps in the *contact-initiation messages* during the contact-initiation part and the Hopping Period Alignment and Adjustment part. C uses the timestamps to estimate its clock drift. According to the estimation, C decides the next time to run the HOPERAA algorithm. C also adjust its hopping period according to the estimation to deal with its clock drift and thus avoid sending messages to closed ports.

In the following sections, 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 the* contact-initiation messages *from the client are called* guard ports.

3.2 Contact-Initiation Part

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

Algorithm 1. Algorithm for client C in the initiation stage

$reply \leftarrow$ **false**
T_{send}, T_{arrive}
/* T_{send} and T_{arrive} store the sending time and the arrival time of the first contact-initiation message received by the server. Once their values are given, they will not be changed. */
- sending contact-initiation messages:
while $reply =$ **false do**
$I \leftarrow select(I_i | i \in \{1, 2, \ldots, k\})$
/* randomly select an interval of port numbers */
for all $p_i \in I$ **do**
$timestamp \leftarrow Time_{now}$
$send\langle Init, timestamp\rangle$
/* Send one contact-initiation message to each of the ports in the chosen interval. */
end for
$wait(2\mu + L)$
end while
- receiving reply message:
$receive\langle ReMsg, \lambda, \sigma, timestamp, h_c(t_1), t_2\rangle$
/* timestamp is the sending time of this reply message, which is t_3 in Formula 1. $h_c(t_1)$ is the timestamp of the corresponding contact-initiation message received by the server, and t_2 is the arriving time of the same message. They will be stored later by the client for estimating its clock drift. See Section 3.4 and Lemma 4. */
if $reply =$ **false then**
$reply =$ **true**
if this is the first time of executing contact-initiation procedure **then**
$T_{send} \leftarrow h_c(t_1)$
$T_{arrive} \leftarrow t_2$
end if
execute the HOPERAA algorithm
/* The pseudocode for the HOPERAA algorithm is given in Algorithm 4 */
start sending data
end if

The server divides the range of port numbers into k intervals evenly and 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 open guard port in each interval. Here, we assume that k can divide N. Note that k is a parameter in the system, whose value is known by the server and the client. C sends contact-initiation messages to all the ports in an interval which is randomly chosen. When S receives a contact-initiation message, it replies with the *seed* λ for the pseudorandom function f_ψ and the index σ for computing the next worker port. The server will send this reply message when the next worker port is open which will happen in at most L time units. Since the network may lose messages and open ports can be disabled by the adversary, C may not get the reply from S. To save bandwidth, instead of keeping on sending contact-initiation messages, C will set a timeout for waiting the reply message. The timeout is set to $2\mu + L$ time units, taking into account the message round trip time and the waiting time by the server to send the reply. If C does not receive a reply until it reaches the timeout, it will choose another interval of port numbers and send contact-initiation messages again until it gets the reply. In Section 5, we will 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 Algorithms 1 and 2, respectively.

Algorithm 2. Algorithm for server S in the initiation stage

- receiving contact-initiation message:
$receive\langle Init, timestamp\rangle$
$h_c(t_1) \leftarrow timestamp$
$t_2 \leftarrow Time_{now}$
wait until next worker port p_i opens
$timestamp \leftarrow Time_{now}$
$send\langle ReMsg, \lambda, \sigma, timestamp, h_c(t_1), t_2\rangle$
/* t_2 and $h_c(t_1)$ are sent back to the client to estimate its clock drift. See Section 3.4 and Lemma 4. */

3.3 Sending the Application Data

In this stage, C sends data messages to the worker ports of S. After C gets the reply from the server in the contact-initiation part, C has the seed λ for the pseudorandom function f_ψ to generate the sequence of the worker ports. The open interval

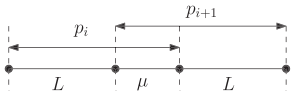

Fig. 1. Worker ports' open interval with overlap.

of the worker ports is $L + \mu$ time units, 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 Fig. 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 $f_\psi(\lambda, \sigma)$. 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 $i \in \mathbb{N}^*$. Note that it might be that the worker port collides with one of the guard ports, and the server can distinguish the contact initiation messages from data messages.

Since there exists delivery latency, some messages that are sent to port p_i (the ith port in the hopping sequence) may arrive when p_i is closed. 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). Messages sent in the interval $[iL - \mu, iL]$ may arrive when p_i is closed. So these message will be sent both to p_i and p_{i+1}. Algorithm 3 shows the pseudocode of client's behaviors in data transmission stage.

Algorithm 3. Algorithm for client C in data transmission stage

$P_{old} \leftarrow f_\psi(\lambda, \sigma)$
$P_{new} \leftarrow f_\psi(\lambda, (\sigma + 1))$
/* P_{old} is the destination port in the current period, and P_{new} is the destination port for the next period. */
- sending data messages
while has more data to send **do**
$send\langle Data, P_{old}\rangle$
if $iL - \mu \le T_c \le iL$ **then**
$send\langle Data, P_{new}\rangle$
end if
end while
- changing the destination port
if $\{T_c = iL\}$ **then**
$P_{old} \leftarrow P_{new}$
$P_{new} \leftarrow f_\psi(\lambda, (\sigma + i + 1))$
end if

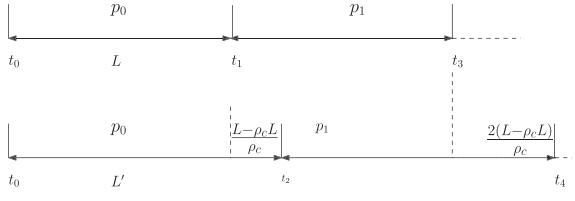

Fig. 2. Client's clock is slower than the server, which means $\rho_c < 1$.

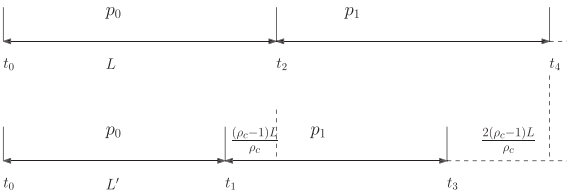

Fig. 3. Client's clock is faster than the server, which means $\rho_c > 1$.

3.4 Adaptive Hopping Period

As mentioned in Section 2, client C has a constant clock drift ρ_c related to the server. It may happen that in the data transmission stage, the hopping time of C will drift apart from the server's. This might cause C to send messages to a port that is already closed or is not opened yet, depending on whether C's clock is slower or faster than S's. Figs. 2 and 3 illustrate the two situations, respectively. In both figures, the server hops the ports every L time units and the client also hops the destination port number every L time units counted by its own clock, which corresponds to L' time units in the server's clock. The deviation between L and L' in both cases (where $\rho_c > 1$ and $\rho_c < 1$) is $|\frac{\rho_c - 1}{\rho_c}|L$. From Figs. 2 and 3, we can see that the deviation of hopping times in the same period grows linearly with the number of periods.

Growth of deviation of hopping times would imply more 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 $\rho_c < 1$, 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 $\frac{\rho_c \Delta}{1-\rho_c}$. If the client's clock is faster than the server's which means $\rho_c > 1$, and we want to keep the offset of the open times counted by the server and the client of a worker port within Δ time units, then the HOPERAA execution interval is $\frac{\rho_c \Delta}{\rho_c - 1}$. However, the client has no idea about its clock drift. We suggest a method that exchanges messages (which are piggybacked) with information about the sending and receiving times (time-stamped with local clock values) between C and S, to estimate the clock drift. This is illustrated in Fig. 4. The procedure of the HOPERAA algorithm is described below, and the pseudocode is given in Algorithm 4.

- The HOPERAA execution interval is initiated to 0. In the contact-initiation part, every contact-initiation message and reply message will be attached with the timestamp of its sending time. The reply message also includes the timestamp $h_c(t_1)$ and the arrival time t_2 of the first contact-initiation message received by the server. When the client receives the reply message, it will store $h_c(t_1)$ and t_2 and keep their values unchanged.

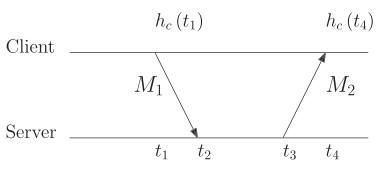

Fig. 4. Messages exchange with timestamps and the associated time points.

- When C executes HOPERAA, it will execute the same operations as in the contact-initiation part, the server will add a timestamp of the sending time to every reply message, say t_3. The client will record the arrival time of the reply, say $h_c(t_4)$. Then, C bounds its clock drift as

$$\frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2 + 2\mu} \leq \rho_c \leq \frac{h_c(t_4) - h_c(t_1)}{t_3 - t_2}. \quad (1)$$

  We use ρ_l and ρ_u to denote the lower and upper bound of ρ_c, respectively. In the analysis section, Lemma 4 will show the correctness of Formula 1; we will also show that every time C estimates its clock drift, it will get a better bound than the one it got from the previous estimation.
- If both the lower bound and the upper bound are greater than one, that is, $1 \leq \rho_l \leq \rho_u$, then adjust L_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 both the lower bound and the upper bound are smaller than one, that is, $\rho_l \leq \rho_u \leq 1$, then adjust L_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 not change L_c, and the HOPERAA execution interval is set to $\min\{\frac{\rho_l \Delta}{1-\rho_l}, \frac{\rho_u \Delta}{\rho_u - 1}\}$.

Algorithm 4. The HOPERAA algorithm

/* This procedure is called in Algorithm 1, and this pseudocode should be plug in the subroutine $receive\langle ReMsg, \lambda, \sigma, timestamp, h_c(t_1), t_2\rangle$ in Algorithm 1. */

- HOPERAA procedure:

$t_{reply} \leftarrow$ the arrival time of $ReMsg$
$\rho_u \leftarrow \frac{t_{reply} - T_{send}}{timestamp - T_{arrive}}$
/* where timestamp is the timestamp included in the contact-initiation reply message $ReMsg$ */
$\rho_l \leftarrow \frac{t_{reply} - T_{send}}{timestamp - T_{arrive} + 2\mu}$
if $1 \leq \rho_l \leq \rho_u \| \rho_l \leq \rho_u \leq 1$ **then**
  $Interval_{HoPerAA} \leftarrow \frac{\rho_u \rho_l \Delta}{\rho_u - \rho_l}$
  if $1 \leq \rho_l \leq \rho_u$ **then**
    $L_c \leftarrow L \cdot \rho_l$
  else
    $L_c \leftarrow L \cdot \rho_u$
  end if
else
  $Interval_{HoPerAA} \leftarrow \min\{\frac{\rho_l \Delta}{1-\rho_l}, \frac{\rho_u \Delta}{\rho_u - 1}\}$
end if
Call Algorithm 1 at time $(Time_{now} + Interval_{HoPerAA})$

Before C adjusts L_c, it has to know whether its clock rate is faster or slower than the server's, otherwise it has no idea whether to shorten L_c or prolong L_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 L_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's, then the client can never know whether its clock rate is faster or slower than the server. But since the bounds of the drift improve monotonically, (cf. Section 5) the HOPERAA execution interval keeps growing (exponentially, cf. Section 6) with the number of HOPERAA executions. This means that the client does not have to do the alignment of the hopping time (which is HOPERAA execution) frequently. The message and time overhead involved in the HOPERAA executions will be amortized within the big HOPERAA execution intervals.

4 Supporting Multiple Clients

The extension to multiple clients per server is based on a simple idea: since each client considers the server's clock as the reference clock, it can interact with the server independently of the other clients. For scalability reasons it is desirable that the server has more than one worker ports open in each time period (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.

As the name also suggests, the BIGWHEEL algorithm, aiming at meeting the aforementioned goals, functions as the Big Wheel rides at amusement parks: clients queue for the next available compartment. Here 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 detail below.

Algorithm 5. Algorithm for server S using multiple hopping sequences.

Buffer B stores reply messages that are waiting to be sent.

- receiving contact-initiation message:

$receive\langle Init, timestamp\rangle$
$h_c(t_1) \leftarrow timestamp$
$t_2 \leftarrow Time_{now}$
/* p_j^i is the ith port number in hopping sequence j. Suppose its open time is the closest to the current time. */
$\lambda_j \leftarrow$ seed for hopping sequence j
$\sigma \leftarrow$ the corresponding index value for p_j^i in hopping sequence j put the reply message $\langle ReMsg, \lambda_j, \sigma, timestamp, h_c(t_1), t_2\rangle$ into buffer B

- sending reply messages:

whenever a new worker port is opened
send all the reply messages in buffer B to the corresponding clients
Clear B

Consider a simple big wheel with one compartment, which corresponds to the situation that the server only opens worker ports according to one hopping sequence. In this setting, since every client uses the same pseudorandom 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 client's waiting time by at most L time units. In order to afford more

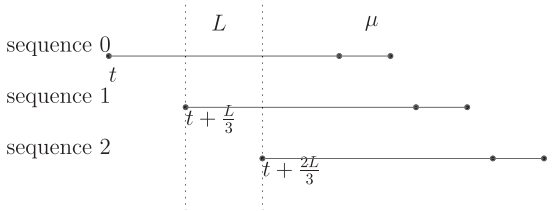

Fig. 5. The open intervals of port p_0^i, p_1^i, and p_2^i.

clients and also decrease the maximum waiting time for a client, the server will open worker ports according to multiple hopping sequences. These sequences can be generated by the same pseudorandom function but with different seeds. For example, sequence i uses seed λ_i. Suppose there are $W \geq 2$ values for λ (i.e., we have W compartments in the big wheel). It means that S supports W hopping sequences. Let us use p_j^i to denote the ith worker port in the jth port number sequence, where $0 \leq j \leq W-1$. 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 $t_i + \frac{jL}{W}$. The open interval of every worker port is still $L + \mu$. Fig. 5 shows the situation when $W = 3$ 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 opening time of a worker port, including the seed for the corresponding sequence. A pseudocode is shown in Algorithm 5. By using multiple port number sequences, the maximum waiting time for a client in the contact-initiation part can be decreased to $2\mu + \frac{L}{W}$ time units.

5 Analysis of the Protocol

We start with some auxiliary definitions which 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 is received by the server.
- A *contact-initiation trial* is a trial by a client to get the server's reply in the contact-initiation part. It begins when a client randomly chooses an interval of the port space and sends a contact-initiation message to each of the ports in that interval and ends when the client receives a reply message from the server or reaches a time-out of waiting.
- We say that the adversary launches a *blind attack* if the adversary arbitrarily chooses and attacks Q ports of the server simultaneously.
- If the adversary knows the ports that are currently used by the server, it will send malicious messages to attack those ports directly. We say that the adversary launches a *directed attack*.

First, we analyze the contact-initiation part of the protocol. Since the guard ports cannot be fixed, the server has to hop guard ports but in a range 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 the 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, $I_i = \{p_j | i\frac{N}{k} \leq j \leq (i+1)\frac{N}{k} - 1\}$, $i = 0, 1, 2, \ldots, k-1$. 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 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, we say this *contact-initiation trial* fails. We will show how to bound this probability and give a corresponding experimental result in Section 6.

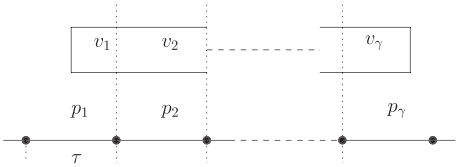

Fig. 6. Arrival duration covers γ changing periods of guard ports.

Lemma 1. *If the adversary launches a blind attack, the probability that it disables the guard port in the interval chosen by the client in the contact-initiation part 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 $1 - (\frac{1}{e})^F$, where $F = \frac{N-Q}{N}$ is the fraction of nondisabled ports .*

Proof. 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 Fig. 6, the arrival duration of those V messages may cover γ changing periods of the guard ports. We use p_i, $1 \leq i \leq \gamma$ to denote the guard port for period i, and use v_i to denote the number of contact-initiation messages that arrive at the server side 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

$$Pr[\text{trial fails}] = \Pi_{i=1}^{\gamma} Pr[p_i \text{does not receive the message}]$$
$$= \Pi_{i=1}^{\gamma}(Pr_d + Pr_l - Pr_d \cdot Pr_l),$$

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

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

Using the method of Lagrange multipliers, we know that $Pr[\text{trial fails}]$ has the maximum value when $v_1 = v_2 = \cdots = v_\gamma = \frac{V}{\gamma}$. 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 $(1-\frac{1}{x})^x$ 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

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

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

□

Corollary 1. *The expectation of the number of contact-initiation trials is at most $\frac{1}{1-(\frac{1}{e})^F}$. The expectation of the number of contact messages used in the contact-initiation part by one client is at most $\frac{N}{k}\cdot\frac{1}{1-(\frac{1}{e})^F}$, 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 since that some client's clock may be faster than the server, and can send messages to a worker port before it is opened, as a result, the adversary could get the worker port number before the corresponding port is opened. However, the adversary cannot get the port number more than Δ time units earlier than the port's opening time, since 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 cannot launch a directed attack to an open worker port of the protocol in this paper.*

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 the reference clock, and consider that the client sends message M_1 at time t_1 with the timestamp $h_c(t_1)$ and the message is received by the server at time t_2. Consider also that the server sends later one message M_2 at time t_3 with timestamp t_3, which is received by the client at time t_4 corresponds to time $h_c(t_4)$ according to the client's clock. Then, we have*

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

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

Proof. Consider Fig. 4. According to the clock drift definition, we have

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

where d_1 and d_2 are the delivery latencies of M_1 and M_2, respectively. Since $0 \le d_1 + d_2 \le 2\mu$, 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 the client C has repeated the HOPERAA algorithm several times. In our protocol, the client 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 HOPERAA execution. Hence, the upper and lower bounds of ρ_c will converge to the real value of ρ_c as the execution progresses.

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

Proof. Suppose the client's clock drift is ρ_c. Then, the client uses $\frac{1}{\rho_c}$ time units to count 1 time unit, i.e. the difference is $|1 - \frac{1}{\rho_c}|$. So if the client wants to keep hopping times not drifting Δ time units away from the server's, the HOPERAA execution interval should be $\frac{\Delta}{|1-\frac{1}{\rho_c}|}$ which equals to $\frac{\rho_c\Delta}{|1-\rho_c|}$. Since $\rho_l \le \rho_c \le \rho_u$, we have

$$\frac{\rho_c\Delta}{|1-\rho_c|} \ge \min\left\{\frac{\rho_l\Delta}{|1-\rho_l|}, \frac{\rho_u\Delta}{|1-\rho_u|}\right\}.$$

The client uses $\min\{\frac{\rho_l\Delta}{|1-\rho_l|}, \frac{\rho_u\Delta}{|1-\rho_u|}\}$ as the HOPERAA execution interval, so the client's hopping times will not drift away from the server's Δ time units.

If the client knows that its clock drift is bigger than 1, which means $\rho_u \ge \rho_l \ge 1$, it will change the hopping period to $L \cdot \rho_l$, which is $\frac{\rho_l}{\rho_c}L$ time units according to the server's clock. Then, the difference between the length of the server's hopping period and the length of the client's hopping period is $(1 - \frac{\rho_l}{\rho_c})L$. So the HOPERAA execution interval should be $\frac{\Delta}{(1-\frac{\rho_l}{\rho_c})L}L \cdot \rho_l$ which equals to $\frac{\rho_l\Delta}{1-\frac{\rho_l}{\rho_c}}$, and the client uses

$$\frac{\rho_u\rho_l\Delta}{\rho_u - \rho_l}$$

as the HOPERAA execution interval. Since $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l} \le \frac{\rho_l\Delta}{1-\frac{\rho_l}{\rho_c}}$, the difference of the hopping times of the client will not drift Δ time units away from the server's.

If the client knows that its clock drift is smaller than 1, which means $1 \ge \rho_u \ge \rho_l$, it will change the hopping period to $L \cdot \rho_u$. Then, the difference between the length of the server's hopping period and the length of the client's hopping period is $(\frac{\rho_u}{\rho_c} - 1)L$. So the HOPERAA execution interval should be

$$\frac{\Delta}{(\frac{\rho_u}{\rho_c} - 1)L} L \cdot \rho_u$$

which equals to $\frac{\rho_u\Delta}{\frac{\rho_u}{\rho_c}-1}$, and the client uses $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l}$ as the HOPERAA execution interval. Since $\frac{\rho_u\rho_l\Delta}{\rho_u-\rho_l} \le \frac{\rho_u\Delta}{\frac{\rho_u}{\rho_c}-1}$, the difference of the hopping times of the client will not drift Δ time units away from the server's. □

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

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

$$N - N\left(1 - \frac{1}{N}\right)^W \le E[w] \le N - N\left(1 - \frac{1}{N}\right)^{2W},$$

where N is the size of the port space.

Proof. Given a specific time point, the probability that port p_i is opened by a specific hopping sequence is $\frac{1}{N}$. Since each sequence opens ports independently, the probability that port p_i is not opened by any hopping sequence is $(1-\frac{1}{N})^W$. Let A_i, $i \in \{0, 1, \ldots, N-1\}$ denote the event that port p_i is open at a specific time point. The expectation of the number of worker ports being open at the same time is

$$E[w] = \sum_{i=0}^{N-1} Pr[A_i] = N - N\left(1 - \frac{1}{N}\right)^W.$$

Remember that the open intervals of the old worker port and the new worker port in a sequence have an overlap, hence the number of sequences for which server opens worker ports can be regarded as at least W and at most $2W$. Hence, we have

$$N - N\left(1 - \frac{1}{N}\right)^W \le E[w] \le N - N\left(1 - \frac{1}{N}\right)^{2W}.$$

□

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

$$1 - \frac{\binom{N-w}{Q}}{\binom{N}{Q}},$$

where N is the size of the port number space.

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

In the BIGWHEEL algorithm, one communication port might be kept open continuously by different hopping sequences or even by the same sequence. We study the situation that a specific port is kept open continuously by multiple hopping sequences. For the sake of the simplicity of analysis, we assume that there is no overlap between the open intervals of two neighbor work ports in the same hopping sequence, and the open interval of each worker port is L time units.

Suppose at time $t = 0$ one sequence opens port p as the worker port. According to the BIGWHEEL algorithm, for every time $t = i\frac{L}{W}, i \in \mathbb{Z}^+$, there is one sequence opens a worker port. It is observed that in order to keep port p continuously open as a worker port, at least one of the worker ports that open at time $t = i\frac{L}{W}, i \in \{1, 2, \ldots, W\}$, should be p. Based on this observation, if a specific port is open as a worker port, then we can compute the probability that this port is kept open continuously for a given length of time.

Lemma 8. *In the BIGWHEEL algorithm, suppose that a specific port, say port* p*, is open as a worker port, the probability that* p *is open continuously for $\mathbb{L}$ time units, $\mathbb{L} > L$ or more, is at most $R(W, \lfloor\frac{W\mathbb{L}}{L}\rfloor, \frac{1}{N})$, where $R(x, y, z)$ is the following recursive function (where x and y are positive integers and $z \in (0, 1)$)*

$$R(x, y, z) = \begin{cases} 1, & y < x, \\ z\sum_{i=1}^{x} R(x, y-i) \cdot (1-z)^{i-1}, & y \ge x. \end{cases} \quad (2)$$

We call the result of $R(W, \lfloor\frac{W\mathbb{L}}{L}\rfloor, \frac{1}{N})$ Continuous Open Probability.

Proof. Without loss of generality, suppose port p is open as a worker port at time $t = 0$, in order to prevent p from being kept open continuously longer than L time units, all the worker ports that open at time $t = i\frac{L}{W}$, $i \in \{1, 2, \ldots, W\}$, should not be p. During $\mathbb{L}$ time units, there are $\lfloor\frac{\mathbb{L}}{\frac{L}{W}}\rfloor = \lfloor\frac{W\mathbb{L}}{L}\rfloor$ worker ports open. We can order these worker ports according to their open times and form a sequence

$$\mathrm{S} = \left\{p_{\frac{L}{W}}, p_{\frac{2L}{W}}, \ldots, p_{\frac{\lfloor\frac{W\mathbb{L}}{L}\rfloor L}{W}}\right\}.$$

In order to keep port p open for $\mathbb{L}$ time units, there must not exit consecutive W ports that are not p in sequence S; in other words, p cannot be open for $\mathbb{L}$ time units, iff in sequence S there exist at least W ports that are not p. Now, the computation of the corresponding probability is the same as the computation of the reliability of a *consecutive-k-out-of-n:F system*. A consecutive-k-out-of-n:F system consists of n linearly ordered components and the system fails if and only if at least k consecutive components fail. The reliability of a such system can be

TABLE 1
Continuous Open Probability

W	$\mathbb{L} = 3L$	$\mathbb{L} = 4L$	$\mathbb{L} = 5L$
2	$3.9998E-12$	$5.0000E-16$	$6.0005E-20$
5	$3.4995E-11$	$7.0013E-15$	$1.2610E-18$
7	$8.3982E-11$	$2.1008E-14$	$4.6268E-18$

computed using a recursive function $R(k,n,p)$ [27] defined as Function 2, where p is the probability that a component does not fail (assuming that every component fails independently and they have the same fail probability). Here, p is the probability that a worker port is port p, which is $\frac{1}{N}$. In Function 2, we use x, y, z as the parameters to prevent reusing of the terms. □

Since Function 2 is a recursive function, it is not obvious to see the relation between the result and the parameters. In Table 1, we show the values of continuous open probability under different settings of parameters. We choose $N = 10{,}000$, meaning the probability that a sequence open a specific port as the worker port is $\frac{1}{10{,}000}$. Table 1 shows that bigger continuous open time $\mathbb{L}$ leads to smaller continuous open probability; also having more hopping sequences leads to higher continuous open probability for a specific $\mathbb{L}$, but the probability is still with very small.

5.1 Overhead Induced by the Mechanism

The overhead of the proposed method consists of message and time complexity in the contact-initiation part, time spent for executing HOPERAA, packets lost due to clock drift. These three kinds of overhead will be studied in three experiments in Section 6. The first two types of overhead have been studied in this section. When a client executes the HOPERAA algorithm, it performs the same operations as in the contact-initiation part. Hence, following Corollary 1 the expected message overhead of HOPERAA is also $\frac{N}{k} \cdot \frac{1}{1-(\frac{1}{r})^r}$ messages 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 amortized overhead becomes smaller in the course of the execution.

6 Experimental Study

To further study the properties of our protocol, we basically conduct three experiments. These experiments validate some of the analytical results and give complementary measures that are not included in the analytical evaluation due to the subtle and complex relations of different parameters. In particular, we show

- The average number of contact-initiation trails that a client has to do under different parameter settings, which conforms to the estimation given in the analysis section.
- The growth of HOPERAA execution interval, which conforms to the algorithm for estimating the client's clock drift.
- The message overhead for initializing the communication can be amortized within a long time scale due to the growth of the HOPERAA execution interval.

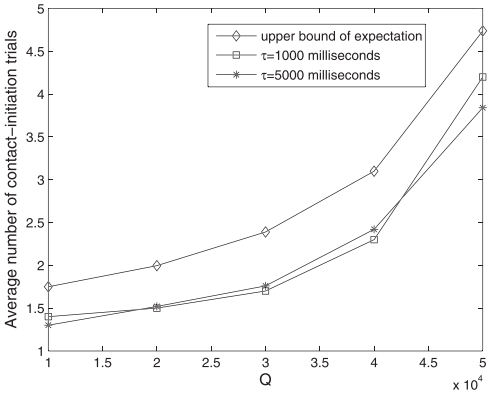

Fig. 7. The average number of contact-initiation trials in one contact initiation part, where $\tau = \{1{,}000, 5{,}000\}$ milliseconds, and client sending rate is 1 message per millisecond

- The messages lost due to the clock drifts can be controlled by adjusting parameters in the protocol.

In the experiments, we assume that the application that uses the proposed port hopping mechanism uses UDP as transmission protocol. On setting up the experiments, we follow the system model described in Section 2.

The first set of experiments simulate the *contact-initiation part*. The experiment is done using two 3 GHz Intel Pentium 4 machines, one acts as the server, and the other one acts as the client. There is no packet loss in the network during transmission. We choose $N = 65{,}536$, and $k = 64$, which translates to 64 intervals in the port space, each interval having 1,024 ports that can be used. We vary the strength of the adversary from $Q = 10{,}000$ to $Q = 50{,}000$.[1] The changing period of the guard ports is assigned to 1,000 and 5,000 milliseconds, meaning that $\tau = \{1{,}000, 5{,}000\}$. For each parameters setting of Q and τ, we let the client perform 50 repetitions of the contact-initiation part, and then record the number of trials of each contact-initiation part. We compute the average number of trials that a client has to perform over all these contact-initiation phases.

Fig. 7 shows both the experimental outcome and the upper bound of the expectation computed in Corollary 1. It is observed that the average number of trials grows with Q, but we can see that even for $Q = 50{,}000$, the average number of trials during the contact-initiation part is still not high (4.2 trials when $\tau = 1{,}000$ milliseconds).

Regarding the average time spent in the contact-initiation part, we take the situation of $Q = 20{,}000$ as an example, where the average number of trials is around 1.5. Since the client needs about one second to send the contact-initiation messages in each trial, and then waits $2\mu + L = 1{,}200$ ms for the reply, the average time spent in this part is about 3.3 seconds. In our experiment, in each trial the client sends 1,024 contact-initiation messages, each having 40 bytes, so the average bandwidth consumed by the client is about 145 kbps.

1. In our experiments, the incoming bandwidth of the server is 10 Mbps. In order to simulate that the adversary can disable, e.g., 50,000 ports completely without congesting the network, we modify Iptables (it is like a firewall) of the server, and drop the legitimate packets to the ports of, e.g., from 1 to 50,000. So only the legitimate packets with the destination port numbers that are not covered by the filtering rules of Iptables can be delivered to the application layer.

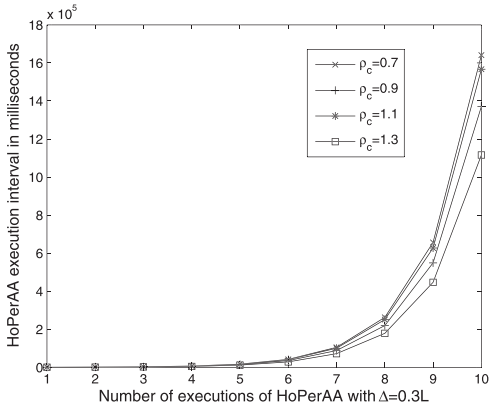

Fig. 8. The length of HOPERAA execution interval grows with the number of HOPERAA executions.

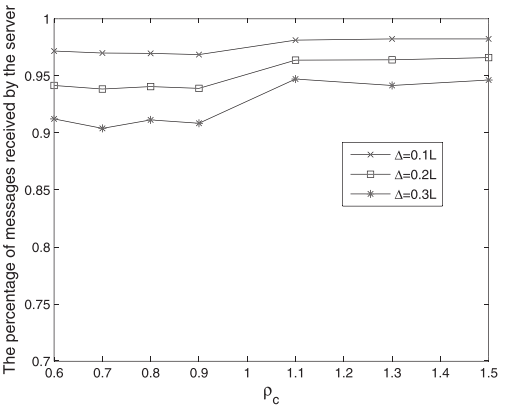

Fig. 10. The receiving percentage of the server; μ is set to 40 milliseconds.

In the second set of experiments we study how the HOPERAA execution interval grows (i.e., the protocol overhead decreases) with the number of executions of the HOPERAA algorithm under different values of clock drift. In the experiment, $\Delta = 0.3L$ and $\rho_c \in \{0.7, 0.9, 1.1, 1.3\}$. As shown in Fig. 8, in most cases, the client will know whether its clock rate is faster or slower than the server's after executing HOPERAA three times, then it will adjust its hopping period. In Fig. 8, we can see that the HOPERAA execution interval grows exponentially with the number of HOPERAA executions. In particular, after eight executions the client can keep sending data messages for more than five minutes, and after 10 executions the client can keep sending data messages for almost half an hour. So the message overhead for initializing communications and aligning hopping periods is amortized within this long time scale.

The last set of experiments study the effectiveness of HOPERAA with respect to the *receiving percentage* which is the percentage of data messages received by the server during data transmission parts. We choose $Q = 0$ (i.e., the receiving percentage is only affected by HOPERAA); the clock drift of the client is $\rho_c \in \{0.6, 0.7, 0.8, 0.9, 1.1, 1.3, 1.5\}$; Δ and μ are chosen as $\Delta \in \{0.1L, 0.2L, 0.3L\}$ and $\mu = \{100, 40\}$ ms. For each value combination of ρ_c 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 Figs. 9 and 10. We can see that the percentage of messages received is very high (above 95 percent) when $\rho_c < 1$ and $\tau = 100$ ms; when $\rho_c > 1$ and $\tau = 40$ ms, the percentage of messages received is also very high (around 95 percent). The lowest receiving percentage in Fig. 9 is close to 90 percent, while the receiving percentage for all the cases in Fig. 10 is above 90 percent.

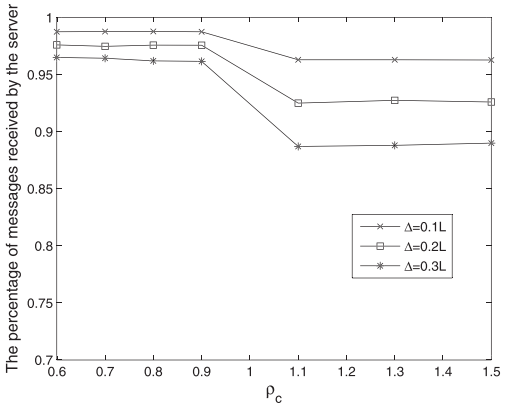

Fig. 9. The receiving percentage of the server; μ is set to 100 milliseconds.

It is observed that the receiving percentage is decreased when Δ is increased. This because bigger Δ means to bigger deviation between the hopping times of the client and the server, which leads to more lost messages during transmission. Actually, the expected receiving percentage is $(1 - \frac{\Delta}{2L})$, e.g., if $\Delta = 0.3L$, then the expected receiving percentage is 85 percent. From Figs. 9 and 10, we can see the experiment result is better than what is expected. The reason for getting better performance is that the client uses $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ as the HOPERAA execution interval. From the proof of Lemma 5, it is shown that this value is smaller than the expected one which is $\frac{\rho_c \rho_l \Delta}{\rho_c - \rho_l}$ for $\rho_c > 1$ and $\frac{\rho_c \rho_u \Delta}{\rho_u - \rho_c}$ for $\rho_c < 1$. 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.

As shown in Fig. 9, when $\rho_c < 1$ the above phenomenon is even more prominent. This is because the client uses Formula 1 to compute the upper and lower bounds of its clock drift and the results are influenced by the message delivery latency bound. In the experiment, the message delivery latency bound μ is much bigger than the actual message delivery latency (the latter is approximately 25 ms in these experiments—recall that μ was set to 100 ms), which results in that ρ_u is closer to ρ_c than ρ_l does. Hence, the ratio between the HOPERAA execution interval $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ and $\frac{\rho_c \rho_u \Delta}{\rho_u - \rho_c}$ is smaller than that between $\frac{\rho_l \rho_u \Delta}{\rho_u - \rho_l}$ and $\frac{\rho_c \rho_l \Delta}{\rho_c - \rho_l}$, which causes the receiving percentage for $\rho_c < 1$ to be higher than the respective one for $\rho_c > 1$.

As shown in Fig. 10, when μ is set to 40 ms which is close to the actual deliver latency, then the receiving percentage for $\rho_c > 1$ is higher than that for $\rho_c < 1$. This is because ρ_l is closer to ρ_c than ρ_u when they are computed using Formula 1.

7 Discussion

When the proposed method is deployed in the Internet, several practical issues have to be addressed. First, choosing

the value of μ is not easy, since the network latency may vary a lot for different sources and destinations. According to the study on Internet round trip time, such as by CAIDA [28], in the implementation, the value of μ that can dominate most cases is approximate $500\ \mathrm{ms}$. However, the value of μ may vary for different applications. For example, the server may almost always serve the clients in the same autonomous system, or in a situation of content distribution network, the servers may serve nearby clients. In such situations, the application can choose a better value (which is smaller than $500\ \mathrm{ms}$) for μ. The value of μ can also be changed dynamically. For example, if our method is used with TCP, then the estimated RTT for TCP can be directly used for setting the value of μ. In the paper, to keep the presentation more focused, we do not include this option. Note that, in some situations (such as flash crowd) the network latency might be bigger than the upper bound, however, the corresponding influence on the clock drift estimation will keep decreasing as mentioned in the analysis.

When the proposed method is used with TCP, the throughput can be affected by the congestion control mechanism in TCP. When the client resumes data transmission after executing HOPERAA, unlike UDP, the TCP transmission will begin with the slow start phase, meaning the congestion window size of TCP will again increase from $1MSS$. However, the influence to throughput is largely amortized, since the frequency of HOPERAA executions drops dramatically, due to the quick convergence of the clock drift estimation. If it is desired to achieve even higher throughput, a deployment option would be to let TCP keep the congestion window size unchanged when the client resynchronizes with the server, i.e., when HOPERAA is invoked.

8 Conclusions

In this work, we investigate application-level protection against DoS attacks. More specifically, supporting port-hopping is investigated in the presence of timing uncertainty and for enabling multiparty communications. We present 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 multiparty communications with port-hopping, an algorithm is presented for a server to support port hopping with many clients, without the server needing to keep state for each client individually. A main conclusion is that it is possible to employ the port-hopping method in multiparty applications in a scalable way. The method does not induce any need for group synchronization which would have raised scalability issues, but instead employs a simple interface of the server with each client. 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. Another main conclusion is that the adaptive method can work under timing uncertainty and specifically fixed clock drifts.

An interesting issue to investigate further is to address variable clock drifts and variable hopping frequencies as well.

9 List of Notations

- C: Client C.
- $h_c(t)$: clock value of client C, when server's clock value is t.
- E: the *exposure delay*. The time it takes for the adversary to get open ports information and get ready to launch the directed attack to the ports.
- f_ψ: the pseudorandom function to generate the hopping sequence(s).
- L: is the length of the server's hopping period.
- N: size of the port number space.
- I: port interval in the port number space.
- k: number of intervals in the port number space.
- Q: the maximum number of ports that the adversary can attack simultaneously.
- Δ: is maximum allowed value of the deviation between the hopping times of the server and the client.
- μ: the maximum message delivery latency.
- τ: the length of the changing period of the guard ports.
- L_c: the length of the hopping period of client C.
- ρ_c: clock drift of client C.
- ρ_l: the lower bound of the client's clock drift.
- ρ_u: the upper bound of the client's clock drift.
- λ: the seed used by the pseudorandom function to generate the hopping sequence.
- σ: the integer used by the pseudorandom function to generate a port number of a specific index in the hopping sequence.
- W: number of hopping sequences used in the BIGWHEEL mechanism.
- w: number of worker ports open simultaneously in the BIGWHEEL mechanism.

Acknowledgments

A preliminary version of this paper was published in the proceedings of 27th IEEE International Symposium on Reliable Distributed Systems (SRDS), 2008 [1]. The research leading to these results has received funding from the Swedish Civil Contingencies Agency (MSB) and from European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 257007.

References

[1] Z. Fu, M. Papatriantafilou, and P. Tsigas, "Mitigating Distributed Denial of Service Attacks in Multiparty Applications in the Presence of Clock Drifts," *Proc. IEEE Int'l Symp. Reliable Distributed Systems (SRDS)*, Oct. 2008.

[2] CERT Advisory CA-1997-28 IP Denial-of-Service Attacks, http://www.cert.org/advisories/ca-1997-28.html, 2010.

[3] K. Argyraki and D.R. Cheriton, "Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks," *Proc. Ann. Conf. USENIX Ann. Technical Conf. (ATEC '05)*, p. 10, 2005.

[4] R. Mahajan, S.M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network," *ACM SIGCOMM Computer Comm. Rev.*, vol. 32, no. 3, pp. 62-73, 2002.

[5] D. Dean, M. Franklin, and A. Stubblefield, "An Algebraic Approach to IP Traceback," *ACM Trans. Information and System Security*, vol. 5, no. 2, pp. 119-137, 2002.

[6] D.X. Song and A. Perrig, "Advanced and Authenticated Marking Schemes for IP Traceback," *Proc. IEEE INFOCOM*, vol. 2, pp. 878-886, 2001.

[7] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," *ACM SIGCOMM Computer Comm. Rev.,* vol. 30, no. 4, pp. 295-306, 2000.
[8] X. Liu, X. Yang, and Y. Lu, "To Filter or to Authorize: Network-Layer DoS Defense against Multimillion-node Botnets," *Proc. SIGCOMM,* pp. 195-206, 2008.
[9] A.D. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure Overlay Services," *ACM SIGCOMM Computer Comm. Rev.,* vol. 32, no. 4, pp. 61-72, 2002.
[10] D.G. Andersen, "Mayday: Distributed Filtering for Internet Services," *Proc. Fourth Conf. USENIX Symp. Internet Technologies and Systems (USITS '03),* p. 3, 2003.
[11] X. Fu and J. Crowcroft, "GONE: An Infrastructure Overlay for Resilient DoS-Limiting Networking," *Proc. Int'l Workshop Network and Operating Systems Support for Digital Audio and Video (NOSSDAV),* 2006.
[12] A. Stavrou and A.D. Keromytis, "Countering Dos Attacks with Stateless Multipath Overlays," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS),* pp. 249-259, 2005.
[13] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing Internet Denial of Service with Capabilities," *Proc. Workshop Hot Topics in Networks (HotNets-II),* Nov. 2003.
[14] A. Yaar, A. Perrig, and D. Song, "SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks," *Proc. IEEE Symp. Security and Privacy,* pp. 130-143, 2004.
[15] X. Yang, D. Wetherall, and T. Anderson, "A DoS-Limiting Network Architecture," *Proc. ACM SIGCOMM,* Aug. 2005.
[16] X. Liu, X. Yang, and Y. Xia, "NetFence: Preventing Internet Denial of Service from Inside Out," *Proc. SIGCOMM,* pp. 255-266, 2010.
[17] J. Mirkovic and P. Reiher, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms," *ACM SIGCOMM Computer Comm. Rev.,* vol. 34, no. 2, pp. 39-53, 2004.
[18] G. Badishi, A. Herzberg, and I. Keidar, "Keeping Denial-of-Service Attackers in the Dark," *IEEE Trans. Dependable and Secure Computing,* vol. 4, no. 3, pp. 191-204, July-Sept. 2007.
[19] Spread Spectrum Scene, http://sss-mag.com/ss.html, 2011.
[20] A. Lempel and H. Greenberger, "Families of Sequences with Optimal Hamming Correlation Properties," *IEEE Trans. Information Theory,* vol. IT-20, no. 1, pp. 90-94, Jan. 1974.
[21] G. Ge, R. Fuji-Hara, and Y. Miao, "Further Combinatorial Constructions for Optimal Frequency-Hopping Sequences," *J. Combinatorial Theory Series A,* vol. 113, no. 8, pp. 1699-1718, 2006.
[22] Y.M. Ryoh Fuji-Hara and M. Mishima, "Optimal Frequency Hopping Sequences: A Combinatorial Approach," *IEEE Trans. Information Theory,* vol. 50, no. 10, pp. 2408-2420, Oct. 2004.
[23] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems," *ACM Computing Survey,* vol. 39, no. 1, p. 3, 2007.
[24] K. Hari and T. Dohi, "Sensitivity Analysis of Random Port Hopping," *Proc. Seventh Int'l Conf. Ubiquitous Intelligence Computing and Seventh Int'l Conf. Autonomic and Trusted Computing (UIC/ATC),* pp. 316-321, Oct. 2010.
[25] H. Lee and V. Thing, "Port Hopping for Resilient Networks," *Proc. IEEE 60th Vehicular Technology Conf. (VTC2004-Fall),* vol. 5, pp. 3291-3295, 2004.
[26] M. Srivatsa, A. Iyengar, J. Yin, and L. Liu, "A Client-Transparent Approach to Defend against Denial of Service Attacks," *Proc. IEEE 25th Symp. Reliable Distributed Systems (SRDS '06),* pp. 61-70, 2006.
[27] F. Hwang, "Fast Solutions for Consecutive-k-out-of-n: F System," *IEEE Trans. Reliability,* vol. R-31, no. 5, pp. 447-448, Dec. 1982.
[28] B. Huffak, D. Plummer, D. Moore, and k. Claffy, "Topology Discovery by Active Probing," *Proc. Symp. Applications and the Internet Workshops (SAINT),* http://portal.acm.org/citation.cfm?id=580055.829312, pp. 90-96, 2002.

Zhang Fu received the BSc degree in computer science from BeiHang University, China, and the MSc degree from Royal Institute of Technology, Sweden, in 2005 and 2007, respectively. Currently, he is working toward the PhD degree in the Department of Computer Science and Engineering at Chalmers University of Technology. His research topic concerns with secure and robust communication protocols. His research interests also include online algorithms and distributed algorithms.

Marina Papatriantafilou received the PhD degree from the Department of Computer Engineering and Informatics, University of Patras, Greece. Currently, she is working as an associate professor at the Department of Computer Science and Engineering, Chalmers University of Technology, Sweden. She has also worked at the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam and at the Max-Planck Institute for Computer Science (MPII) Saarbruecken, Germany. Her research interests include distributed and multiprocessor computing, including synchronization, communication/coordination, with emphasis in robustness, fault-tolerance and dynamic aspects, as well as applications in communication, transportation, electricity networks adaptiveness, robustness and security.

Philippas Tsigas received the BSc degree in mathematics from the University of Patras, Greece, and the PhD degree in computer engineering and informatics from the same University, in 1994. Currently, he is working as a professor in the Department of Computer Science and Engineering at Chalmers University of Technology. From 1993 to 1994, he was with the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam. From 1995 to 1997, he was with the Max-Planck Institute for Computer Science, Saarbrucken, Germany. He joined Chalmers University of Technology in 1997. He is the cofounder and the head of the Distributed Computing and Systems research group at Chalmers. He is the initiator and one of the designers of NOBLE, a library of nonblocking data structures. His research interests include parallel and distributed computing, parallel and distributed systems, and information visualization.