Distributed Malware Analysis Scheduling

Rodrigo Rubira Branco [1]
Gabriel Negreira Barbosa [2]

[1, 2] *Qualys Vulnerability & Malware Research Labs (VMRL)*
[1] *rbranco@qualys.com*
[2] *gbarbosa@qualys.com*

Abstract

Automation of malware analysis is a complex challenge faced by researchers due to the growing number of unique malware samples. For this automation to succeed, the algorithm used in the scheduling decisions must be reliable and efficient to better use the available resources. In this paper we discuss our distributed approach in details, showing the reasoning behind each part composing the scheduler, the distribution of jobs and the analysis results.

1. Introduction

Badware is software that fundamentally disregards a user's choice about how his or her computer or network connection will be used. A badware designed for criminal, political, and/or mischievous purposes is known as malware. [1]

There are many types of malwares reaching a high amount of computers around the world [2], and to make this scenario even worse, there exist lots of distinct samples in the wild. In order to implement counter-measures for such a threat, a deep understanding about each sample is needed; thus, this huge scenario requires and automated malware analysis system.

Automated malware analysis platforms, due to the high amount of samples, often relies on distributed computing to effectively process all the available data. So, the distribution of analysis tasks among network nodes is the core component for the consistency and performance of such platforms: this component is called scheduler.

Qualys has a system for distributed malware analysis, developed for the security research community, in which the processing machines may be anywhere over the internet [3]. It supports multiple machines, with distinct processing power. The scheduler internals description of this system is the goal of this paper.

This paper is organized as follows. In Section 2, there is an overview of the developed scheduler architecture, and the algorithms behind it are described in Section 3. Finally, Section 4 has a conclusion of the work.

2. Scheduler Overview

The developed scheduler is a software component and can be divided in three parts:

- Scheduler Election
- Scheduler Dispatcher
- Scheduler Decision

Every machine in the architecture runs the scheduler component, but not all the parts. The machines may be in the lab's network or not. The main advantage to have outside machines is to allow external entities to contribute with processing power.

The Election part is executed in all machines of the system and is responsible to elect only one of them to be the Main Scheduler.

Main Scheduler is the name of the machine that, besides the Election part, runs also the Decision. Additionally, it runs a small portion of the Dispatcher. There is only one Main Scheduler in the whole system and it is inside the lab's network.

The Decision part is responsible for discovering pending malware analysis tasks and delegating them to the other machines for processing. This communication with the other machines is made through a network-based protocol discussed in Section

4, and is the only portion of the Dispatcher that runs in the Main Scheduler. So, the Main Scheduler runs the Election and Decision parts completely, and only the communication protocol portion of the Dispatcher.

With the exception of the Main Scheduler, all other machines execute, besides the Election, the fully Dispatcher part. Dispatcher is the part of the scheduler responsible for taking tasks from the Main Scheduler and processing them. Additionally, it implements a network-based protocol for all network interactions, such as reporting its memory status and its available slots to the Main Scheduler. A machine that runs the Dispatcher part is also called "Dispatcher".

The Main Scheduler does not run the Dispatcher part completely because:

- It is an important machine, and the high amount of processing due to the malware analysis would increase the crashing probability.
- If it was overloaded, the delegations could be delayed in a way that powerful machines would be idle even with tasks pending for execution. This would decrease the processing power of the whole lab.

In the most common scenario, the most powerful machines are Dispatchers to improve the processing power of the lab, and the weakest one is the Main Scheduler.

Sections 3, 4 and 5 discuss, respectively, the three scheduler parts: Scheduler Election, Scheduler Dispatcher and Scheduler Decision. Finally, Section 6 has the conclusion of this work.

3. Scheduler Election

It is used a distributed leader election algorithm, based on the Bully Algorithm [4], to determine the most appropriate machine to be the main scheduler. The mains aspects of such an algorithm for the architecture are:

- To elect a leader with no dependencies on existing resources such as central servers
- To recover after a scheduler failure (re-election)
- To have one, and only one, scheduler running at a time (uniqueness)
- To identify scheduler failures (watchdog-like behavior)

The main scheduler is always in the lab's local network to make the accesses to the central database optimized and to restrict the firewall rules denying access to such a resource for external hosts.

As soon as a machine is inserted in the lab's local network and got an IP address from the DHCP server, it joins the processing infrastructure and may be elected the leader.

As stated in Section 2, by default the least powerful machine in the lab is the main scheduler, but a priority definition in a per-machine basis allows a 'forced election' of a specific machine for such a function.

To identify the processing power, an index was developed in a way that lower numbers have high processing power. It is calculated as the elapsed time, in milliseconds, of some pre-defined procedure, such as a Fibonacci recursive algorithm for n numbers of the sequence.

The priority number follows a widely-used convention in which lower numbers means more priority.

To define the leader, it is first verified the priority. If more than one machine remains candidate, the processing power index is considered. If it still was not possible to determine the leader, the machine with the lower IP number is elected the leader.

The leader election algorithm is overviewed in Section 3.1 and more formally defined in Section 3.2.

3.1. Leader Election Algorithm Overview

The implemented leader election algorithm is based on Bully Algorithm [4]. In such a variation, the base algorithm was modified in the following aspects:

- Use of TCP to send messages for all available addresses in the network, simulating a “reliable” broadcast with no message loss.
- Priority and processing power utilization to elect the leader.
- Added a feature for the leader to know all the available hosts in the network dynamically.

The Bully variation works on networks that may loss or corrupt messages. Additionally, it automatically recognizes new machines as soon as they are plugged in the lab network.

The main messages are “leader request”, “leader request confirm”, “victory”, “leader alive", and “ACK”.

The “leader request” is broadcasted (conventional IP-based broadcast) to the network for a first trial. It has the priority and processing power values for the destination hosts to compare and reply only if they fit more in the leader position.

If the “leader request” message is broadcasted and no responses arrive, this node is a strong leader candidate (it would be the real leader in a network with no message loss). In such a case where no responses arrived, the host send a TCP message, called “leader request confirm”, to all IPs in the lab’s network also with its priority and processing power values. This is a network-consuming event, and this is the reason behind the first message being sent through a conventional IP-based broadcast: to eliminate as much leader candidates as possible to reduce the number of such confirmations. This message bypasses the message loss restriction with IP-based broadcast approaches.

After the “leader request confirm”, if no replies arrive, this node is the leader and the “victory” message is sent to all machines in the lab. Here, there is another chance for other hosts to “bully” the almost-elected leader. In case of arriving only “ACK” messages, the IP addresses of such sources will be added to the dispatchers list for malware analysis delegation purposes and external machines will be notified. If some “abort” message arrives, this machine will not define itself as the leader and neither does take and delegate tasks; additionally, the machine that aborted it will initiate a new leader election procedure.

In order to avoid unnecessary elections and to solve technical-based leader problems (crashes, network link down problems, etc), it is often sent a “leader alive” message. If such a message was not received as expected for n consecutive times, a new election process is started by the hosts that detected such state.

In the proposed approach, a message is considered as lost if they do not arrive in T seconds. When there is no time constraints to elect the leader for the malware lab scenario, such time may be tuned for high values, increasing the effectiveness of the algorithm.

3.3. Leader Election Algorithm Details

This section describes deeply the implemented leader election algorithm that, as stated in Section 3.1, is a variation of the Bully algorithm. For all the procedures below, some global variables are used and they are initialized with default values:

```
leader := 0.0.0.0; isLeader := false; priority := READ(); processingIndex := PROCESSING_INDEX(); T := WAIT_TIME(); localNodes := {}; node_address := GET_NODE_IP();
```

The main leader election procedure runs in an infinite loop by all nodes and is defined as:

```
MAIN_LEADER_ELECTION():
  while (isLeader = true) do
    broadcast “leader alive”;
    sleep T1;

  while (leader = 0.0.0.0 and isLeader = false) then
    broadcast (“leader request”, priority, processingIndex);
    if (received a message M = (“leader request: abort”, “leader”) then
      leader := GET_SOURCE(M);
      break;
    else if (received a message M = “leader request: abort”) then
      sleep T2;
      break;

    tcp_broadcast (“leader request confirm”, priority, processingIndex);
    if (received a message M = (“leader request confirm: abort”, “leader”) then
      leader := GET_SOURCE(M);
      break;
    else if (received a message M = “leader request confirm: abort”) then
      sleep T2;
      break;

    tcp_broadcast (“victory”, priority, processingIndex);
    if (received a message M = (“victory: abort”, “leader”) then
      leader := GET_SOURCE(M);
      break;
    else if (received a message M = “victory: abort”) then
      sleep T2;
      break;
    for all messages M = “ACK” received do
      localNodes := localNodes U GET_SOURCE(M);

    isLeader = true;
    for each node N outside the lab’s network do
```

```
        send N "victory";

    if (isLeader = false and leader != 0.0.0.0) then
        sleep T3;
```

In the procedures above, there are three main messages: "leader request", "leader request confirm" and "victory". For each of such messages, there are functions with the corresponding handling procedure that are called whenever the related message arrives:

```
LEADER_REQUEST_HANDLER(M):
    (M_message, M_priority, M_processingIndex) := M;
    M_source := GET_SOURCE(M);

    if (priority < M_priority or (priority = M_priority and processingIndex > M_processingIndex) or (priority = M_priority and processingIndex = M_processingIndex and node_address < M_source)) then
        if (isLeader = true) then
            send M_source ("leader request: abort", "leader");
            localNodes := localNodes U M_source;
        else
            send M_source "leader request: abort";


LEADER_REQUEST_CONFIRM_HANDLER(M):
    (M_message, M_priority, M_processingIndex) := M;
    M_source := GET_SOURCE(M);

    if (priority < M_priority or (priority = M_priority and processingIndex > M_processingIndex) or (priority = M_priority and processingIndex = M_processingIndex and node_address < M_source)) then
        if (isLeader = true) then
            send M_source ("leader request confirm: abort", "leader");
            localNodes := localNodes U M_source;
        else
            send M_source "leader request confirm: abort";


VICTORY_HANDLER(M):
    (M_message, M_priority, M_processingIndex) := M;
    M_source := GET_SOURCE(M);

    if (this node is outside lab's network) then
        leader := M_source;
        return;

    if (priority < M_priority or (priority = M_priority and processingIndex > M_processingIndex) or (priority = M_priority and processingIndex = M_processingIndex and node_address < M_source)) then
        if (isLeader = true) then
            send M_source ("victory: abort", "leader");
            isLeader := false;
        else
            send M_source "victory: abort";
    else
        if (isLeader = true) then
            isLeader := false;
            leader := 0.0.0.0;
        leader := M_source;
        send M_source "ACK";
```

New machines, when plugged in the lab network, start a new election because they do not know the current leader. This behavior does not happen for external machines, because they are aware about the current leader as "victory" messages arrives, as stated in Section 4.

Often, the elected leader will broadcast "leader alive" messages. This will let local network machines to know about leader availability troubles and, if n consecutive "leader alive" messages do not arrive as expected, a new election process is initialized. The nodes, as soon as they receive such messages, they will reply, allowing the leader to know all the nodes that are alive in the local network. This behavior also happens for external machines, but they do not start an election if n expected messages do not arrive because they cannot be elected as leaders.

The described algorithm relied on the following pseudo-instructions and functions:

- broadcast: A conventional IP broadcast.
- tcp_broadcast: For each IP in the network range, send a TCP message.
- send <IP> <message>: Send a TCP message <message> to <IP>.

- GET_SOURCE(M): Through the network packet related to message M, extract the source address field of IP header.
- READ(): Get the information (in this case, the priority) through some way: as a parameter, standard input, configuration file, etc.
- PROCESSING_INDEX(): Returns processing index. For example, by the taking the time elapsed to execute a Fibonacci recursive algorithm.
- GET_NODE_IP(): Returns the current node's IP address.
- sleep <N>: keeps the application sleeping for N seconds;
- External IPs: The external machine addresses are obtained by the central database. This creates a dependency on a central point, but improves the security because all external partners have to be pre-defined, avoiding attackers to insert bogus machines to compromise the lab. Additionally, the central database is used by the leader to take and delegate tasks; so, if the central database was unavailable, the system would be down and the leader election algorithm would not be useful.

4. Scheduler Dispatcher

Dispatcher has two distinct functions depending on who is executing it. The main scheduler's dispatcher is responsible for delegating malware analysis tasks to the other machines. Such other machines, called dispatchers, execute the other aspect of the Scheduler Dispatcher, that take tasks from the main scheduler and processes them. As stated before, this last behavior may be present in machines outside the lab's network, allowing the world-wide community to contribute with the analysis by sharing processing entities.

If dispatchers encounter errors while communicating with the main scheduler, some amount of time is wait before trying to re-send the message. This procedure will happen until the message was correctly arrived or a "victory" message was received from a new scheduler; in the last case, the message will be re-sent to it.

To improve the scheduling policies, it was developed a network-based protocol in which the dispatchers and the main scheduler may exchange information such as processing power and available slots.

All messages of this protocol rely on TCP; thus, each message has only one destination. In the case where a node needs to send the same message to three distinct targets, it will need three distinct connections, one for each destination node, to send the same message. In the case where a node needs to send many messages related to the same context for the same destination, a unique TCP connection is used.

The dispatcher protocol is very simple and counts on network-based file systems. The main messages are:

- get machine memory
  - Direction: main → dispatcher
  - Description: Used to get the machine available memory (total in the system).
- report machine memory [total in bytes]
  - Direction: dispatcher → main
  - Description: Used by the dispatcher to report the total available memory on the system [in bytes].
- get machine memory free
  - Direction: main → dispatcher
  - Description: Used to get the machine free memory.
- report machine memory free [total in bytes]
  - Direction: dispatcher → main
  - Description: Used to report the total of free memory in the system.
- get machine vm list
  - Direction: main → dispatcher
  - Description: Used by the main_sched to get the list of vm information. It is expected to receive a list of report machine vm commands.
- report machine vm vm_X
  - Direction: dispatcher → main
  - Description: Used by the dispatcher to report the vm IDs it is using. It is good to know the number of VM's defined for this machine in the configuration file for scheduling purposes.
- get machine vm vm_X status
  - Direction: main → dispatcher

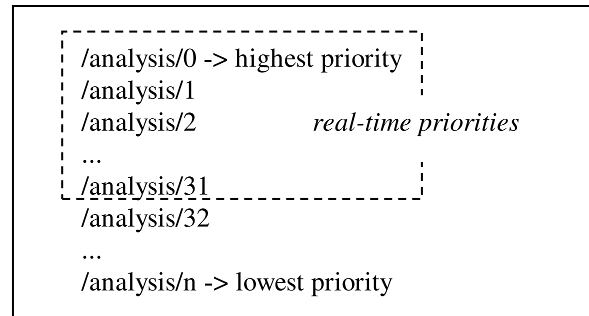
- Description: Used by the main_sched to get information about specific VM ID.
- report machine vm vm_X status [int value with bit definitions]
  - Direction: dispatcher → main
  - Description: Used by the dispatcher to send information regarding a specific VM_ID: IN_USE (true/false) to show if the machine is currently in use by an analysis, and ON_REVERT (true/false) to report if the machine is currently in revert process.
- get machine vm vm_X memory
  - Direction: main → dispatcher
  - Description: Used by the main_sched to get memory information regarding the VM ID.
- report machine vm vm_X memory [amount in bytes]
  - Direction: dispatcher → main
  - Description: Used by the dispatcher to report the amount in bytes for the VM memory.

5. Scheduler Decision

The machines in the lab are dynamically added and removed due to the flexible architecture: it is simply needed to, respectively, plug and unplug the machines in the lab network. To delegate malware analysis tasks to Dispatchers, the Decision first needs an algorithm to constantly discover the machines current plugged in the lab: this algorithm is also implemented by the Election part, but is only executed at the elected machine (Main Scheduler), as described in Section 3.

To better delegate tasks to Dispatchers, the Decision uses its reduced Dispatcher part, which has only the communication protocol known by Dispatchers (Section 4), to get some configurations of all other machines (Dispatchers) in the lab, such as VM memory status and the availability of plugins execution.

The main scheduler runs inotify interfaces for each of the subdirectories inside the root analysis directory. This directory hierarchy is a remote file system its definition is in the lab configuration file. The subdirectories are named by numbers that represent the analysis priority. The following diagram considers "/analysis" as the root analysis directory:



By default all samples go to the directory 32 (normal priority). Priorities less then or equal to 31 are called real-time priorities and are discussed later.

During the initialization, the main scheduler first defines the inotify interfaces to check the remote directories to see if they already have files. In case of a crash, a first execution or a new elected machine acting as the main scheduler, the directories may have pending files to be processed, and some troubles may raise: if a file in process notification is received, the main scheduler restarts the analysis process for such a sample: it is better to analyze twice than miss an analysis. Then, the main scheduler contacts all the dispatchers to get the number of supported VMs, plugins and number of cores.

Each machine in the laboratory is defined as a group of VMs, with each VM running on that machine been part of that group.

Every machine receives three different queues:

- Idle queue: Only in use when the machine is doing nothing (this is the case when the report command will show IN_USE = false → Important: Not all VMs in the architecture are going to be reverted and every time we have computing capacity we want to already boot the vm for the next sample.
- current: The current queue. Samples here are going to be executed in order in the group, or in parallel if multiple VMs that support the sample are available.
- Next: For example, the group has two VMs:
  - VM1: Plugin 1

- VM2: Plugin 1 and 2

A sample runs in the VM1 against plugin1, and after that it needs to be executed by plugin2 also. After running in the plugin1, the sample goes to the next queue. Other samples that need to run against the plugin1 and are already in the current queue will run first to avoid cases where samples are going to run on multiple VMs, slowing down the whole system: execute where it is able to, report whatever it gets and then later on execute again in other VM as necessary. The exception here is for the real-time priorities (0-2 by default, but defined in compile-time): the samples always go back to the current queue, meaning they will be favorable for execution.

To define in which machine to run each sample, the scheduler basically creates in the initialization:

- The groups and its associated running queues
- The list of samples and its associated

The priority ranges (0-n) are actually defined in a higher value for a logical reason. The system defines 64 run queues and to select a run queue to insert the specific sample, it divides the sample priority by 4 (thus, the sample normal priorities are internally between 160 and 223 and real-time priorities between 128 to 159). Thus, the directory value (0-n) is added to 128 to discover the real priority and it is then divided by 4 to discover the run queue. In each queue, the samples are NOT sorted by its priorities.

An array with the head of each queue exists and the queue itself is organized as a doubly linked list of sample structures. The aforementioned array also has an associated bit vector that is used to identifying nonempty run queues.

Thus, the scheduling has three functions:

- runq_add() → place a new sample in the tail of a run queue.
- runq_del() → remove a sample from the head of a run queue (and update the array).
- runq_choose() → select a sample from the queue (when there are available resources in a group). It does so by:

1) Acquiring a lock (to manipulate the array, queue, bit vector and group current queues).
2) Locating a nonempty run queue -> find the first nonzero bit in the bit vector. If the bitvector is zero, there is nothing to do.
3) If there is a nonempty queue, remove the first sample from the queue (runq_del()).
4) If the queue is now empty as result of the removal, reset the appropriate bit in the bit vector.
5) Put the sample in the selected group current queue.

Every time a sample is define to run, a THRESOLD configurable (in the config file) timer is defined for the sample. After the THRESHOLD the sample is reinserted in the queue for analysis (assuming a timeout, error or whatsoever).

6. Conclusion and Future Work

Distributed architectures for automated malware analysis allow a deep analysis because of the high amount processing power. Additionally, it can scale together with the growing number of unique samples.

In such a decentralized approach, the scheduler is a central point for the analysis to succeed, because it is responsible to maximize the available resources utilization.

This paper showed a robust scheduler model with components that allows malware research systems to scale with the growing number of malwares and to execute heavy analysis algorithms to achieve deep results. Additionally, the main protocols and algorithms involved in the scheduling were detailed.

In spite of all the benefits, there is always place for improvements and the presented m,alware analysis system is constantly being optimized. Regarding the distributed leader election algorithm, there is still space for research of centralized options because there is a dependency on the central database by the main scheduler and all nodes currently have to directly access it to process samples and using such machine may improve the scheduler performance.

7. References

[1] Stop Badware Project. Referemce: http://www.stopbadware.org. Last visited: July/2011.

[2] Microsoft Security Intelligence Report, vol8. Reference: http://www.microsoft.com/security/about/sir.aspx. Last visited: July/2011.

[3] Branco, R. and Shamir, U. Architecture for Automation of Malware Analysis. Malware2010, 2010.

[4] Hector Garcia-Molina, Elections in a Distributed Computing System, IEEE Transactions on Computers, Vol. C-31, No. 1, January (1982) 48-59