Monito: A scalable, extensible and robust monitoring and test-deployment solution for planetary scaled network.

ADRIEN DUFFY--COISSARD
Monito: A scalable, extensible and robust monitoring and test-deployment solution for planetary scaled network.

Adrien Duffy-coissard
adrien.duffy@gmail.com
2016

Under the supervision of:

KTH: Johan Montelius
DICE: David Röhr
Abstract

Development of a monitoring and testing solution using gossip and leader election protocols. This solution has to ensure scalability, extensibility, over nodes distributed in a planetary-scaled network, up to several thousands of monitored units. The reliability and quality of the network should not limit the performances of the framework. The goals are also to limit the costs and additions of components and computer resources, without interfering with the services delivered by the monitored structure. The project is limited to data collection, test distribution, protocols and architecture description, but does not provide any solutions for human interaction interfaces.

Dedicated to my grand father Jacques, Armand and Alain who all passed away during the writing of this report. You are part of my life.
Acknowledgements

Thanks to all the people without who this report would not have existed.

Thanks to my supervisors, Johan Montelius at KTH, and David Röhr at DICE, for taking the time to lead me through this research, with calm, patience and wisdom.

Thanks to all my team at DICE, Erman Döser and Johan Kamb for their help, Johan Dalborg for his kind support! Thanks to DICE for providing me with everything I needed to go through my research, and Alexander Hassoon and Peter Kjellberg for opening this master thesis subject to me. I would definitely like to thanks David H., Adrian I., Ovidiu C., Mihai S. and Erik P. for their social support, it was a real pleasure to work close to them!

Special thanks to my parents and grand parents for their life-time support to help me become who I am today.

Thanks to all my teachers who gave me the capacities to learn, with a special mention to Christian Daujeard, whose passion, knowledge and commitment were a great inspiration and helped me find my way.

Thanks to David Hewlett for his great role as R. McKay in Stargate, who inspired me as a child to follow the path of Computer Science. He is the biggest influence that turned my passions toward science.
Contents

1 Introduction ........................................ 6
   1.1 Context ........................................ 7
   1.2 Recommended courses ................................ 7
   1.3 Terms definitions ................................ 7
   1.4 Objectives and requirements ....................... 8

2 Previous work ..................................... 9
   2.1 Softwares ....................................... 9
      2.1.1 Datadog .................................... 9
      2.1.2 Sensu ...................................... 9
      2.1.3 Assimilation project ......................... 9
   2.2 Scientific papers ................................ 10
      2.2.1 Astrobale .................................. 10
      2.2.2 GEMS ...................................... 11
      2.2.3 Tree-based and hybrid solutions .......... 11

3 Dividing the data flow ............................ 12

4 Leader Election over a gossip membership discovery 12
   4.1 Gossip membership discovery ..................... 12
      4.1.1 Different approaches ....................... 14
      4.1.2 The SWIM protocol .......................... 14
      4.1.3 How to adapt the SWIM protocol to our case 16
      4.1.4 Conclusion .................................. 19
   4.2 Reliable Broadcast ................................ 20
      4.2.1 Problem and previous work ................. 20
      4.2.2 A solution using SWIM ....................... 20
      4.2.3 Evaluation of the solution .................. 21
      4.2.4 Conclusion .................................. 22
   4.3 Leader election protocol ......................... 23
      4.3.1 A multiple leaders election ............... 24
      4.3.2 Evaluation of the solution ................. 27
      4.4 Section Conclusion ............................ 28

5 The use of leaders to ensure data flow ........... 29
   5.1 The leader as a buffer .......................... 30
   5.2 Evaluation ...................................... 32
   5.3 Conclusion ...................................... 35

6 The data flow, base of a test deployment protocol 35
   6.1 Objectives ...................................... 35
   6.2 A test deployment without external component .... 35
   6.3 A more secure CDN-based protocol ................ 40
   6.4 Conclusion ...................................... 45

7 Scaling the election leader protocol ............... 46
   7.1 Division in groups, the bootstrap node as a key node 46
   7.2 Avoiding topology problems ..................... 47

8 Future work ....................................... 49

9 Conclusion ....................................... 51
1 Introduction

From companies to research centers, the need for computer resources is increasing through time, leading to a growing number of architectures built in a distributed way. More servers, more machines, in order to have more storage, more computer processing power. Being more efficient to deliver a service, to solve a problem, or simply to maintain a strong company infrastructure is essential. But increasing the number of computer units is not only a problem when it comes to create the architecture, it is also a problem when it comes to test and maintain those units running, and be able to detect any kind of failures, not only for the unit itself but also for the environment surrounding the unit: infrastructure, network. These notions of test deployment and monitoring are a real challenge, and a wide range of solutions can be used, from using third-party softwares, to develop a specific solution. The challenge is different, depending on the use of the infrastructure to monitor, its size, the importance of its reliability and many others points. In the studied case, the challenge is to create a way of deploying tests and monitor an infrastructure that can grow up to tenth of thousands of units, change of topology, with every units and the network being not reliable. Yet, the solution must be robust to crashes at the units level but also on its own system level. The costs of adding the solution on top of the monitored infrastructure must be as limited as possible. In this paper, we will see that the goal of the project is to provide a service, pluggable, that takes care of carrying tests from a user to machines, and bringing back data to the user. In order to do that, a study of the existing monitoring and test deployment systems must be done, pointing to possible limitations of the existing solutions, and trying to build a solution which gets rid of as many limitations as possible. We will then try to conceive an architecture for that solution, and see what services that architecture can offer, which limitations are present and try to theoretically anticipate problems, explain how the solution deals with them, before testing it on simulated or real environment. A lot of research has been done in that field, and this paper will focus on research about monitoring protocols, postponing the test deployment concern, that we will try to plug to the monitoring architecture. Whenever it comes to scalable solutions, whose maintaining cost would not explode with the number of units to monitor, the ideas are to aggregate the information, and use basically tree-based or gossip structures to provide aggregation, as we will see in 2. Most of those solutions are quite scalable with the number of nodes, but not really with the number of data exchanged, in case we don’t want to aggregate the data. The structure is also always complex to maintain, especially the tree-based ones, and some points in the solutions were whether bottlenecks, whether non-robust elements that would not fit our problem. This is why an extended research has to be done. We will see some of these existing solutions before discussing about a new solution. This paper will focus on protocol description on will not offer any concrete example of how to implement those protocols, the main interest not remaining in such concerns.
1.1 Context
This paper has been written for a master thesis, as a part of the Degree in Information and Communication Technology, with the specialty in Distributed system, at KTH, Stockholm. The writing of the paper follows a 5 month internship in an international company, where the solution discussed in this paper has been implemented and tested. The company, offering online services through their products, was in need for a solution of monitoring and test deployment, able to complete their existing solutions. With nodes being deployed and terminated in a dynamic way, and with a planetary scale infrastructure, the requirements of the monitoring and test deployment solution, defined in 1.4, are not allowing trivial solutions. This is why searching for the existing solutions was a real need, in order to extract the strengths and weaknesses of each solution, before creating our own alternative.

1.2 Recommended courses
Strong notions in computer sciences are essential to the correct understanding of this paper, including network, security and distributed systems knowledge. Course such as KTH ID2201, “Distributed Systems, Basic Course”, KTH IK2206 “Internet Security and Privacy” are recommended to ensure a perfect understanding of the discussions in this paper. No software development notions are required.

1.3 Terms definitions
In this paper, we will use a vocabulary that needs to be defined:

- **A node**: will refer to a computer unit needed to be monitored, that can be a server or any device with a capacity to run programs.

- **Neighbor**: the neighbor of a node is another node to which the first node can communicate with its current knowledge of the system.

- **The system**: will represent the entire set of nodes that need to be monitored

- **The network**: will mainly stands for the structure supporting communication between the nodes. Can also be used instead of the word “system” in certain context.
1.4 Objectives and requirements

The final objective of the solution is to provide a service being able to deploy tests on a system, and receive data from this system. This solution does not include the test creation, nor the data processing for being used by a user. It focuses on the architecture and protocols responsible for the flow of data from (and to) the user, from (and to) the nodes in the system. The solution should meet the following requirements:

- The complexity of the solution at a connection level should remain as low as possible, even with a growing number of nodes in the system.
- The load on the nodes of the system and the network should remain low, and not grow rapidly with the number of nodes in the system.
- The solution should be able to run in a non reliable network, being able to provide part or all of its services when handling network failures, until a certain point.
- The solution should be adapted to a changing system, with nodes being able to join or leave the network (following a crash or normally).
- The costs of the solution should be bounded, by limiting the use of additional and dedicated components.
- The solution should get close to work in real-time.
- Data must be able to be communicated without aggregation if needed.
- Network topology should not prevent the solution to run normally. We make the assumption that a nodes can communicate with another node as long as the node possesses the information needed for the connection to be set. We will see through this paper how this point verifies in our case.
2 Previous work

Several monitoring solutions have been described through different scientific papers, and some monitoring solutions already exist, from open-source softwares to world-wild known companies providing monitoring services.

2.1 Softwares

2.1.1 Datadog

Datadog, a monitoring and analytic platform [1], provides a well known service giving the possibility to companies to be able to keep track of the state of their servers. The architecture presented in [10] follows a point-bridge-point connection, with the monitored nodes pushing data to a forwarder. The latter receives data from multiple nodes, buffers the data until a certain load is reached, and then sends it to the Datadog servers for data processing. If this solution seems to work well, (with Datadog offering to monitor thousands of nodes in the pricing section), it cannot be adapted straightforward to our case, since it only gathers information about specific nodes, without offering the possibility to deploy tests and check internal network behavior functioning. Moreover, using this kind of architecture in our solution comes with a high cost, due to the forwarders that must be deployed on top of the already existing nodes to monitor. As our solution is supposed to be robust to failures, having to rest upon the forwarders components means that they must be able to handle failure. No solutions are offered concerning this point, with Datadog probably ensuring forwarders availability by adding a layer of reliability to the machine, replicating the data or balancing the incoming connections. Even if this is an assumption, the robustness of a machine always raises costs problems, that we want to avoid in our solution.

2.1.2 Sensu

Sensu, a cloud monitoring service described in [18], uses queues to handle incoming data from the monitored nodes. A server can then read the queue and handle the data as it comes. Several handlers can be processing the data to be more efficient. Queues bring a really interesting solution to our problem, since they are easy to set up, provide mechanisms to avoid loss of data when crashing, and provide scalability by the way the data is pushed/popped from the queue. But scalability is only reached by increasing the number of queues and workers to process the data, which implies dedicating new servers on top of the existing nodes, at a certain cost. While providing an interesting tool, Sensu does not totally satisfy our requirements with the number of connections to the message queues having to grow at the same rate than the number of nodes in the network. We will see if this approach can be used or adapted in our solution in the next sections.

2.1.3 Assimilation project

Assimilation project, [13], based on a ring architecture, with intra and inter subnetwork monitoring. Every node is part of a ring of nodes, called a sub-ring, with the nodes member of those sub-rings monitoring each of their neighbors. In order to connect the nodes together, all the sub-rings are linked together to form a greater ring. This link is done by connecting one node of each sub-ring together. This is building a tree-based structure where a node of the tree is a ring. So we have rings playing the role of internal nodes of the tree, and monitored nodes playing the role of the leaves. The multi ring system makes scalability easy, and prevent too much data to be exchange between nodes that are not on the same network, by configuring rings so that they
are built from nodes in the same network. The problem is to maintain a ring structures in a dynamic infrastructure, where nodes can crash and start at any time. This require a manager to manage all the nodes organization, which is complex, centralized and thus does not scale well. Monitoring is limited in Assimilation project to a simple failure detection, and the problem of gathering more monitored information is not solved, and test deployment seems complex to plug.

We can see that the commercial software or open source projects are bringing important elements to a solution that could fit our requirements. From the bridge-nods structure of Datadog, to the message-queue use of Sensu, we are introduced to components and architecture that could be used in a solution. But there is a need for modification, as those solutions fail to satisfy robustness and scalability at the same time. We will now have a introduction to scientific papers focused on creating monitoring protocols and architecture that scale well.

2.2 Scientific papers

To face scalability problems, some papers present solutions based mainly on two different kind of architectures :

- Horizontal, with gossip protocols or cluster utilization
- Vertical, with hierarchical organization - tree-based structures

Those papers are based on aggregation mechanisms, to reduce the amount of data exchanged, by aggregating the data in different points during its way to the monitoring server. Those aggregation points are usually the key part of the solutions : their properties, number and reliability makes the architectures different from each others.

2.2.1 Astrobale

Astrobale, [20], is an important paper in monitoring solutions, because of its focus on scalability and robustness. The four principles defining Astrobale are its scalability, its flexibility, its robustness and security. With such principles, Astrobale seems to perfectly fit our requirements. However, Astrobale presents two main issues that will prevent it to be an answer to our problem, and those issues are directly linked to the architecture and protocol it uses. Indeed, Astrobale is based on the idea of dividing the nodes into zones, a zone being a set of nodes or a set of zones. Every zone is aggregating the data from its direct children, whether they are other zones or the nodes itself, thanks to aggregation functions. And this raise a problem : how do we define the zones ? Astrobale specifies that the zones must be defined by the user. This is problematic when it comes to scaling, since the user has a key role in the process, and having nodes regularly leaving and joining the network makes the zone definition hard to maintain. Moreover, Astrobale is based on aggregation. If this is the main process that ensures scalability, it seems impossible by concept in Astrobale to store and collect individual data, without seriously impacting the performances of the system, needing to replicate aggregated data for a zone through the nodes of this zone to ensure reliability. This solution does not prove useful to solve our problem, but is perfectly adapted to solve common monitoring problems allowing aggregation and with a system composition not changing frequently.
2.2.2 GEMS

GEMS [16], takes an important place in this thesis, since it presents an interesting way of monitoring data: every node belongs to a group of nodes inside of which it exchanges data using a gossip protocol, piggy-backing the data on the gossip messages. A consensus is reached in the group about the nodes status, in an aggregated form. A second layer can then ask one node in each group about the consensus result, and aggregate all those results to form a view of the state of the entire infrastructure. One of the interesting point of this architecture is the structure of the agent running on each monitored node. This agent is divided in different components:

- A gossip agent providing a communication layer
- A monitoring agent that measure the performances of the node and uses the communication layer to send the data.

This way of splitting the roles in different components results in an extensible agent. We will keep this way of dividing the solution in layers, and see how we can build our solutions by plugging services on top of the others to build Monito. The aggregation and two-layers system provides good scalability, several layers can be added to reach a higher scalability. But again, as we saw in Astrobale, the aggregation of data is the main issue here since it does not allow, again, to exchange non-aggregated data without a huge cost on the volume of data exchanged inside of a group. Moreover, consensus are real challenges to implement in a system that needs consensus in a short amount of time, with network and nodes failures, as we will discuss in the Leader election protocol part.

2.2.3 Tree-based and hybrid solutions

Another paper, [17], presents a solution based on a tree structure. Each leaf is a monitored node, and each internal tree-node aggregate the data from its children. The tree structure is built over the monitored nodes, only with the monitored nodes, except from the root that is the monitoring server. This solutions satisfies our cost limitation requirement, however, the main problem comes from the high load on the root node if no filter or rate limitation is applied to the solution. Maintaining a tree structure with nodes crashing and connecting often is complex, and this complexity along with the high load of data on the root does not provide the solution adapted to our needs.

GANGLIA, [12], is a monitoring solution that takes the best of both tree-based and gossip-based structures. Nodes are organized in clusters, each cluster having a representative node. A tree of point to point connections is built over the cluster representatives to aggregate data. A multicast protocol is implemented inside of each cluster to implement membership discovery and monitoring. This solution provides high scalability since it relies on tree structure over multiple and expandable number of clusters. But it relies on a multicast, that can be complex to implement if nodes are in different location, and that does not scale well since it can flood the network if too many nodes are in the same cluster. The tree nodes are not robust by nature, and then should be avoided to be used.
3 Dividing the data flow

With a better knowledge of the previous work done in the monitoring field, we can now decide on what kind of global architecture we want to achieve. In Monito, we need an architecture scalable, with a potential way of aggregating data, and being able to be robust to node failures and network failures. In the part Previous work, several of the solutions, including Datadog, were using node representatives, or bridge nodes to gather data in a first step. This provide an excellent way of aggregating data, while dividing the data flow across several nodes. However, the solutions were using external components to fulfill this role, leading to a contradiction with our requirements, whether on a robustness or cost point of view. What is needed is then to have the node in the system that are fulfilling the role of bridge node, avoiding to depend on external components. Since nodes are frequently leaving or entering our system, we need to find a solution that allows to have changing bridge nodes. If the nodes are able to change, they should be able to handle failure of nodes as well. It seems that dividing the data flow through intermediate nodes, bridge nodes, is an interesting idea, when those bridge nodes are chosen among the network. But even with bridge nodes, the upper node finally receiving all the data will have to deal with many connections to handle data from the bridge nodes. Direct connection has to be avoided, and several options can be considered, from using a load balancer to divide the flow to several machines, to use Message queues, as seen in Sensu [18]. The interest of message queues is that they provide scalability and reliability: data sent to the message queue can be persisted to handle crashes and recover the data. Moreover, the main node receiving all the data can process it from the message queue, and it also allows several processing nodes to read the data from the message queue, bringing another level of scalability.

Finally, we decide to build our solution on this global scheme: a main server, reading data from a message queue, and bridge nodes in the system sending data to this message queue, data that they collect from all the nodes in the system. With free solutions like RabbitMQ to implement the message queue, the costs of the solutions are almost non-existent, but the message queue still needs to run on a machine with enough resources to handle the message queue at a high scale. With the message queue properties and the potential protocols to chose bridge nodes, we offer robustness, and do not force the aggregation of data, while making it possible at the bridge nodes. We will see later that the capacities of the bridges nodes, and the way they are chosen, will lead us to decide to call them “leaders”. While a bridge node seems to be adapted for a static structure whose data is exchanged through the bridges, our equivalent nodes will be dynamic, and take some decisions that normal nodes would not be able to take. This is why the name “leader” will be used from now until the end of this report, to replace the term of “bridge” node described in this section.

4 Leader Election over a gossip membership discovery

The way of choosing the bridge nodes from our architecture decided in 3 will be the core of our solution. This way of picking special nodes in a set of nodes refers to leader election. We will from now call our bridge nodes “leaders”, and we will try to build a solution to elect them, while satisfying the requirements from 1.4

4.1 Gossip membership discovery

In large distributed systems, having the knowledge of the member composition of the system or network, can be of a great use. If in some cases the set of nodes forming the system can remain identical through time—making possible to easily build the map of the system members—,
most of the distributed systems are nowadays deployed on a high scale and with important node turnover. The turnover implies node leaving and joining the system, in a regular way or following a crash. Maintaining the nodes connected to each others with a changing composition of the system is a challenge, and this is even more challenging when it comes to running the system on a regular network, potentially impacting the communications with message drops, latency and failures. The system is vulnerable to these failures, and the fact of looking for a protocol running at a high scale level would make the use of TCP or other reliable, ordered or error-checked delivery protocol less recommended, since they cost more resources to run. We will see through or solutions that reliable connections can be used at some points, but most of the transmissions will be based on UDP.

A popular solution when it comes to dealing with scalability and members joining and leaving the network is to implement the desire behavior on a top of a gossip protocol. Gossip protocols are based on the idea of propagating, or more precisely disseminating any kind of information in a similar way than a gossip would propagates among humans in a real life environment. A node gives information to a neighbor\(^1\), and the neighbor exchanges this information to another node as well, and so on until all the nodes in the system are aware of the information. While the information propagates, the nodes having already propagated the information can continue to exchange it with another neighbor, making the information spreading faster with time, as a gossip would among humans. This information can be the list of members of the system, or any other kind of data, leading to multiple applications, from network monitoring as seen in [20], to failure detection in [19].

If the protocol seems easy, it requires for a node – to be able to communicate with the neighbors – to have the knowledge of the members in the system. We will call the information contained in a node about the other nodes in the system the “view of the system”. If this situation looks like the initial problem, it is not the case. As explained, the information exchanged by the gossip protocol can be the list of the current members of the system. The nodes can share information and update their view of the system according to the information they receive. But maintaining the entire view of the system updated at each node has a great cost of performances: it does not scale well with an increasing number of nodes, for the memory taken by the whole list or for the time it requires to update the whole view. Some gossip protocols use the approach of the partial view ([5], [6]). Instead of having the entire view of the system, each node keeps track of a subgroup of nodes of the system. The way of obtaining such a view is crucial in the quality of the gossip protocol. The goal is to have:

- The entire system mapped through all the partial views, with redundancy introduced between the partial views
- All the nodes “interconnected”, meaning that two nodes could communicate even if they don’t know each others, if the message they exchange goes through several views

We will focus in the next discussions on the way of exchanging information concerning the membership of nodes of the system. While the base of the protocol remains the same than a regular gossip protocol, and the issues are similar, the naming we will chose for the elements exchanged will be related to the membership discovery vocabulary only. We will first try to see which are the different gossip protocols based on partial views managing for membership discovery. We will then explain the protocol that will be chosen and see how to adapt it to our special scenario.

\(^1\) A neighbor is any other node of the system that the node knows
4.1.1 Different approaches

In order to understand the following section, it is important to know that:

- The steps of the protocol through time are defined as cycles. Even if globally, the protocol is asynchronous, locally, every time new information is exchanged – or an exchanged is initiated resulting in information being exchanged –, we define a cycle. This is used to understand the steps of the protocol.

- An information, here a node identification – is sorted, among the other information contained by a same node, by its age. That means that a value is chosen to sort nodes depending on their order of arrival at every node. For instance, if a node A receives information at a cycle 2 from node B, and at cycle 3 from node C, C will be sorted as younger as B. The age of the node is sent in the information exchanged.

The different solutions for a gossip protocol that we are going to take into account are distinguishable in three points, as detailed in [9]:

- The way the node chooses the neighbor that will be used to exchange information: picking a node among the partial view is done in a specific way, as seen in [9] which retain two relevant ways: random or oldest node.

- The way the information is exchanged with his neighbor, whether in a push – a node will push the information to another node – or in a pushpull - the information is exchanged in both sides –

- The way of selecting the information to exchange: the node should chose between randomly selecting the information to send, or picking it depending on the age of the information.

It is interesting to see in details what are the advantages of each methods, but we will, in our case, chose to follow [5], a discovery membership gossip based protocol, that is introducing a different way of picking the information to send, while working with a third way of exchanging the information: a pull. The decision to chose this protocol compared to others is due to the good results it provides, complying to the goals we are trying to reach. One could argue that this gossip protocol could be chose from another solution: this is indeed the case, and we will see that the way our solution is divided – into layers – makes every layer – or as we call it, service – easy to replace with another solution, as long as the same requirements are fulfilled. It is then important to know that one of the point of this paper is to provide a global solution, whose parts could be changed to fit others requirements needed by the reader.

4.1.2 The SWIM protocol

SWIM [5] is based on detecting the failures of nodes and being able to quickly identify the nodes that are no longer in the network. In that way, it is perfectly adapted to our initial need: being able to identify leaders failures, while providing a partial view of the network in order to use it for different application in Monito. A complete description and evaluation of SWIM is done in [5], but we will summarize the protocol and the advantages it provides for our solution.

SWIM uses a random neighbor selection, and is inspired by a heartbeat protocol to keep track of the nodes in the system. While a heartbeat protocol would require whether a central node to keep a heartbeat to all the nodes, or every node to keep a heartbeat with all the nodes, it is something that cannot scale well, on the first case because it is centralized – a unit has to handle all the protocol – and in the latter because the load on the network would increase quadratically with the number of nodes. The idea of SWIM is to have every node launching a single heartbeat.
process at every gossip cycle toward a random neighbor. So every cycle, a random neighbor is asked for information through a PING. If it answers – a PONG –, the node is alive, and its information can be merged with the heartbeat initiator’s information. If the neighbor does not answer, it is not yet consider as failed, since the network could be responsible for the non arrival of the answer, but this missing answer event triggers a protocol that will lead to a death identification if the node has failed. SWIM, like [9], attaches an age to every node information that is exchanged. This age, while it serves partly the same purposes than the [9] – being able to merge the partial views in a clever way – is different. This difference comes from the fact that a node is described in SWIM with several parameters:

- Its identification as a node, that is customized for every implementation but yet should contain enough information to contact the node.
- The state of the node: A node can be alive, suspected or dead, we will see later the meaning of those states.
- The incarnation number: this number is the key of a clean merge of the partial view, preventing message reordering to impact the detection of failure or the membership service.
- At every node, the number of times the information about a node has been exchanged since the last update of this information is stored, but never exchanged. It impacts the way of selecting the information to exchange.

The three states of a node reduces the impact of network failures such as message drop or latency on the service, that could create false-positive failure detection. Those three states are:

- Alive: corresponds to a node that answers to a PING. An alive node is seen as a correctly running node in the system.
- Suspected: corresponds to a node that seems to fail to communicate, that is potentially dead, but since a failure of the network could be responsible for this supposition, the node is not set as dead yet
- Dead: corresponds to a node that is not answering, even after having taken into account network failures. Such a node is, for the system, crashed or terminated, and is no longer member of the system.

Every node of the system stores and maintains a partial view of the system, this partial view being a list of node descriptors containing the information stated above. Through the ping-pong exchange, a node will merge its partial view with incoming information, with specific rules applied depending on the state and incarnation number of the node information.

The global heartbeat protocol for a given node $X$ is defined by SWIM as:

- Every cycle, define in SWIM by a delta of time $t$, the node randomly picks a neighbor among its list of alive and suspected nodes.
- The node $X$ sends a UDP PING message to the neighbor picked and wait for its answer (all the communications use UDP).
- When the neighbor receives the ping, it selects a subset among its partial view, this subset being a set of a given size of the nodes information that have already been sent the lowest number of times.

See [5] for more details on the given sizes and times chosen in the protocol.
• The neighbor sends this partial view in a PONG message

• Upon receiving the PONG message, the node X merges it with its partial view \(^3\) and the cycle is done.

In case the pinged node is not answering after a given amount of time, a subroutine protocol is launched:

• The node X randomly chooses a small given number of nodes among its partial view of alive and suspected nodes and send them a ping-request

• Those nodes will play the role of bridges\(^4\), trying to forward the ping to the originally pinged node, and forwards the pong answer message back to the node X

This step avoid latency and messages drop to impact the heart beat protocol.

If no answer is received after another given amount of time, the pinged node state in the partial view of the node X is set to Suspected. From now, several things will modify this transitional state:

• A partial view is received and contains information about the suspected node: if the received information is an Alive state and the incarnation number is greater than the local incarnation number, the received information overrides the local one. If the received information is a dead state, it overrides the local one.

• No information overriding the suspected state is received, and no PONG message is received from the pinged node, then after a given amount of time, the node is confirmed dead.

This protocol offers a good scalability, with the different result from [5] showing an infection time \(^5\) that is not increasing with the number of nodes, up to 56 nodes. Moreover, a bad quality of the network is not strongly impacting the membership discovery. However, a false-positive death detection could occur, and this would still be a problem in our case. With SWIM, there is no way for a node confirmed as dead to be alive again on the network. We want to avoid that, our main interest being to be able to detect failures of leaders, while keeping the membership view as accurate as possible. We will see how to adapt the SWIM protocol in order to make possible for a node to rejoin the system even if declared as dead.

4.1.3 How to adapt the SWIM protocol to our case

The main reason a dead node cannot be overridden as alive is the consequence of the protection against message reordering. Incarnation numbers are here to prevent message reordering to false the views by creating a global order on nodes state. This order, the incarnation number of a node, can be modified only by the node itself. Thanks to that number, a node can be unsuspected only if the suspected node received information about its own suspicion, and incremented its incarnation number to prove that it is not dead but alive. Any node receiving the information that the node is alive with a higher incarnation number will unsuspect the node. But now, let’s consider that a node Y suspects a node X. Then the node X is aware of the suspicion, increments its incarnation number. When Y receives information that X is alive with a higher incarnation number, Y will unsuspect it. But if the message from X is stuck for a while, and meanwhile, Y

\(^3\)See SWIM 4.2 part for more details on how to merge the views

\(^4\)One must differentiate the use of bridge as describing the behavior of the node in the present case, and the use of bridges to describe the special nodes of Datadog that we used in the previous section

\(^5\)Time for a node information to reach all the nodes in the network
crashed and is confirmed as dead in the network. \(Y\) will receive the message later, while \(X\) is dead. If it was possible for an Alive state to override a dead state, \(X\) would be consider as alive again, while he is not.

The idea is to create a new state NEW, that is the initial state of every node joining the network. This state is introduced with those rules: NEW overrides a DEAD state if the NEW incarnation number is greater than the DEAD incarnation number. In the other cases, NEW behaves as ALIVE, being overridden by DEAD and SUSPECTED. A change is brought to the DEAD override. While in SWIM, a dead state overrides any incarnation number, in our solution, a DEAD state overrides only if it is greater than or equal to the local state. This prevents an old DEAD state to override a node that would have rejoined the system after its failure. Moreover, to limit the data drop, we limit the size of every message by dividing the pong messages in several messages if the set of nodes information sent is over a given size. This size should be decided depending on the probability of message drop for a given size, which is out of our scope. In our leader protocol, the death of a leader will result in its elimination from the leader set, but a false-positive death will not exclude the node from the network.

We test in a mininet simulation network of 30 hosts, each node exchanging a partial view up to 10 nodes of each state for 1, and 30 for 2, first with 7% of message drop and a latency up to 700 milliseconds. We chose 1.5 seconds to wait for a PING reply, 2 seconds to wait for a bridge node PING reply, and 13 seconds as the time for a node to be confirmed dead. The set of bridge nodes is of size 1. Several measurement have been done, all leading to random variation between 27 and 30 nodes in the network. Since the average of such a measurement would not be relevant, we will discuss about one of the measurement, all the measurements showing the same behavior.

![Figure 1: Number of alive + suspected nodes through time with a network with 7% of message drop since t=0](image)

17
We can see in 1 that some nodes are detected dead as false-positive, around 

\[ t = 25 \]

for instance. This is, as expected, happening after the period of time needed for a node to be detected as dead, this period being:

\[ 1.5s + 13s + 2s \]

, so 16.5 sec, corresponding to the sum of the timers waiting replies from the different steps of the protocol. As we can see, few seconds later, the number of nodes in the system reaches its maximum again thanks to the introduction of the “NEW” state. The figure 2 confirms that false positives are quickly reintroduced as alive, and thus even with an increasing number of nodes in the system. The NEW state offers a new way of maintaining the membership accurate. It is more adapted to our case, while keeping the same scalability and robustness of SWIM.

Another modification to the SWIM protocol is added: in a ping message, the identification of the sender is sent with its incarnation number, to make sure that the communicating node will be merge with the destination’s partial view. This will slightly speed up the process of unsuspicion and then lower the rate of false positive.

To start the gossip protocol, the node joining the network needs to have basic information, so at least some nodes already in the network to start to communicate with. Since no members are robust to failures, the best way to provide these information is from a bootstrap node. This solves the ID creation issue, by allowing each node to generate its own ID, and checking with the bootstrap node if the ID is unique in the system. This bootstrap node will be required to be robust, or at least should always remain with the same address/name so that any node entering the network can contact it to have bootstrap information. This node will run the same protocol than the others nodes, except that it will be a passive member. Any node from the system will push a part of its partial view once in a while as if the bootstrap node was sending a ping before.
That reduces the resources needed for the bootstrap node, and this passive behavior solves a important problem concerning the topology, as we will see later in part 7.2. A node entering the network will contact through a TCP connection the bootstrap node and wait to receive the bootstrap information. The TCP connection ensures that this important step is fulfilled even in case of network failures corrected by TCP. This will also ensure that the connection is made possible from the bootstrap node to any other node, with topologies that would prevent simple UDP to be used, as we will see later.

4.1.4 Conclusion

Thanks to the gossip protocol, we have now introduced in our solution a great way of providing nodes with a certain view over the network, leading to future possibilities to implement a reliable broadcast, and a leader election protocol. This gossip based membership discovery is explained and has been evaluated in the [5]. Avoiding a centralized or heartbeat protocol between every nodes, this solution provides a reasonable load on the network, increasing linearly with the number of nodes in the system. Since the only data sent each cycle is limited to the protocol communications: ping, pong, ping-request, pong-request, one node communicating with no more than one other node at a time every cycle (if we consider that an exchange happens in one cycle, if not it just adds a factor to the linear growth). We have then successfully answered to the requirement of limiting the load on the network at a high scale. The data exchanged being a part of a partial view, in terms of quantity of data exchange and not of unit of exchanges, we stay with a low load of data, depending on the size of the set sent every message. Our solution benefits from SWIM robustness to network failures, while not overloading nodes with the need of TCP connections. The solution is also robust to node failures, and more generally to nodes leaving and joining the network, which makes it great for distributed systems.

However, the use of UDP limits the protocol to run behind the same NAT or at least forces the nodes behind NAT to have a public IP. Protocols to maintain a membership discovery protocol through NAT exist, for instance in Swim through Nat, a course paper from KTH, but requires an overload of resources, or the presence of open nodes accessible by every other nodes. Because of this constraints, it has been decided to limit our scope to a gossip protocol between nodes being able to communicate with UDP. With a solid way of having an view of the system, we are now able to build some useful services in Monito to provide the monitoring and test deployment services.
4.2 Reliable Broadcast

Ensuring a reliable broadcast can prove to be useful in an application. Reliable means that all the nodes part of a group of nodes should receive a message that is broadcasted to the group. We will see in this part why the broadcast problem cannot be solved in the same way as in a regular non-distributed system, then we will introduce the existing work to decide on a solution, before evaluating this solution.

4.2.1 Problem and previous work

In a distributed system with a changing topology, nodes from different local networks can be interconnected. Exchanging information sometimes requires to broadcast a message through all the nodes of the system, and the distribution of nodes can prevent the use of a regular broadcast. While multicast could be a solution, ensuring the reliability of the multicast with nodes entering and leaving the network and network failure is challenging. We are, as for the member discovery solution, still trying to find a solution that can scale, be robust to node and network failures, close to real time, avoiding to add external components while keeping a low load on the network and a low consumption of resources on each nodes.

4.2.2 A solution using SWIM

Thanks to the membership discovery protocol, a node is able to have a partial view of the network. When we look back at the solution used for this membership discovery, it is quite straightforward to realize that the gossip protocol is implementing a broadcast over the network. Indeed, an information of the failure of a node is eventually reaching every node in the network following the dissemination properties of a gossip protocol. The same result occurs for the information of a node joining the network. If this solution fits all the requirements concerning scalability, load on the network, resource usage and robustness, it is not optimal for a fast broadcast, since the dissemination is limited by the cycles required by by SWIM. The other problem concerns memory management: the list of information to gossip and the information itself can be of an important size, buffering it in the same way the node information are buffered in the membership discovery protocol would go against our low-resource utilization policy on the nodes. The idea is then to use the partial views to create a second layer of broadcasting, this time again based on a probabilistic and dissemination approach, but faster than the membership discovery gossiping. While the solutions for a reliable broadcast we introduced are efficient, they are too complex for our case. We will see later that the requirements for the broadcast layer are not as strict as a reliable broadcast should be, for some protocols we will explain are taking into account the fact that the probability for all the nodes to receive a broadcasted message is high enough. According to [6], with some assumptions,

The solution is simple, it consists in broadcasting a message to a subgroup of nodes taken from the partial view. The number of nodes used to broadcast is called the fanout. All the nodes chosen receive the message from the broadcast node, and will repeat the process again. The idea is to cover all the nodes with this protocol of propagation. In order for the algorithm to stop and to avoid to flood the network, it propagates the message only a given number of rounds, called Nrounds, and all the nodes that already received the message and broadcasted it locally are not gonna broadcast it again if they receive it a second time. To be able to identify if a message has already been received, the original broadcaster node needs to identify the message with an id. In this solution, we use the node ID and a counter to identify a message. A first broadcast will start with a counter at 0 and every new message to broadcast from the same node will increase the counter by one. The solution to identify a message at the receiver is to store the last counter
received for each node, and accept a message as new only if the counter of the message is greater than the last counter received from the same sender. If the solution does not seem optimal for scaling, due to the fact that a node stores all the counter/id he receives to be able to identify a new message as new, we can remove the oldest values stored to maintain a reasonable size of the storage structure. An incoming message whose sender’s counter is not stored will be considered as new.

4.2.3 Evaluation of the solution

To evaluate this solution, we try to see how many rounds it takes for all 100 nodes of the system to receive a message with different values of fanout, in a simulation of the broadcast algorithm, considering the fanout to be random nodes list of randomly distributed partial views.

![Graph showing number of rounds and messages needed to broadcast a message to all the group with an increasing fanout.](image)

As we can see in the graph, the bigger the fanout is, the lower is the number of rounds needed to achieve a full delivery. Under a fanout of 5, the nodes do not receive messages because the protocol stops by picking nodes that have already received the message. We now have to decide which value to choose in the protocol. When the fanout reaches 9, the average number of messages decreases since less rounds are needed, in average. To provide a good scalability, we have to avoid flooding the network with too many messages. The maximum number of messages sent in the whole process is $\text{Number of nodes} \times \text{fanout}$, since a node can broadcast a message only once, and the maximum of messages corresponds to all the nodes broadcasting locally once. A fanout
of 9 or 10 seems to be a good solution, since the number of rounds remains quite low, and the number of messages exchanged remains not too high.

We run several tries introducing this time 7% of message drops, and we will see how this algorithm handle network failures.

![Figure 4: Number of rounds needed to broadcast a message to all the group with an increasing fanout with 7% message drop](image)

The results obtained are not surprising: the global scheme remains the same, with only the minimum fanout required to have all the nodes receiving the messages increasing by one, being 6 instead of 5. The number of messages exchanged is not increasing, which is logical since even if the number of rounds increases a bit, 7% of the nodes chosen in the fanout do not receive the message each local broadcast in average, leading to slightly less nodes sending messages every round.

### 4.2.4 Conclusion

We chose to use a broadcast built on the membership discovery protocol, in order to keep meeting the requirements of the global solution. This broadcast mechanism is not provided by any additional components, and provide a good robustness to network failures as seen in 4. The execution of this protocol being fast, the real-time requirement is also satisfied. The overload of the network and the scalability of this protocol might be subject to discussion. As we have seen, the maximum number of messages exchanged can be quite high, and is linearly growing with the number of nodes in the system, except at one point, with a factor equal to the fanout. Broadcast large messages should be avoided, but keeping simple sized messages is not a problem.
To reduce the amount of message simultaneously on the network, a delay can be introduced at each node before broadcasting the message locally. With a random delay, all the nodes would then not broadcast at the same time, preventing potential bottlenecks to be overloaded with messages. More measurements should be taken in order to clearly study the average number of messages exchanged and the real time load of messages on the network. But with this solution, Monito possesses a service to broadcast, and a quite robust one. Knowing the pros and cons of this solution, we can now build others services on top of it in order to reach the goal of this paper, a monitoring and test deployment solution.

4.3 Leader election protocol

The goal of this section is to find a way of electing and maintaining a subset of the nodes as leaders. Those leaders would then fulfill a specific role of leader node, as we wanted to have in order to divide the data flow, reduce the data load on the network and on the message queues by making aggregation possible. With the introduction to both a member discovery service and a probabilistically reliable broadcast service, we have now the tools to create a leader election protocol. As we have seen in the introduction of this section, leader election protocols are applied in an environment where no coordinator has the ability to elect a node among the group. The election protocol must happen by exchanging messages between nodes in the group, so that they elect their leader themselves. The diversity of leader election protocols comes from the diversity of parameters in a group running the protocol. Those parameters defining the group and impacting the leader election are ([3]):

- the network topology: the way the nodes are interconnected will be a key part in the different protocols (e.g. mesh, ring, unidirectional rings)
- the identification of the nodes: in the system, the nodes can have identities or being anonymous. Identities, if they can be sorted, can be an easy way to select a leader in some topologies ([4])
- the knowledge of the network: the nodes in the group can have the knowledge of all the other nodes in the group, or only partial knowledge, or even no knowledge at all.
- the transmission synchronization: asynchronous or synchronous, impacting the possibility of ordering and date messages, and then impacting the election protocol.

According to [8], in a probabilistic leader election, which is what we are looking for, they are three conditions that must be reach with a given high-probability:

- Uniqueness: among all the nodes of the system, there is one and only one node that is aware of being a leader, and this node is non faulty
- Agreement: among all the nodes of the system, all the non-faulty nodes are aware of the leader, and they agree about the identity of this leader.
- Scale: we must be able to bound in time and number of messages a round of the protocol election, without dealing with the system size.

If some solution exist depending on the system parameters, they often require a consensus, and the consensus is challenging to be obtained in a system of interconnected nodes, which are not organized in a particular structure, which can fail, and whose nodes only have knowledge of a partial view of the network. Some solutions like the Paxos algorithm to reach consensus are complex and an agreement can be long to be reached [11].
This definition and conditions required to run a leader election protocol are different from the condition and requirements of the leader election we need. And this is one we will not spend time introducing the different leader election algorithms, that are specific to meet the conditions of a probabilistic leader election. Indeed, having a unique leader as a special node is not relevant, we need several leaders in the group. What we want is to be able to maintain the existence of several leader through time, and thus with nodes and network failures. The unique leader election is then becoming a multiple leader election protocol.

4.3.1 A multiple leaders election

For the purpose of a multiple leader election, we define new conditions:

- **Existence**: among all the nodes of the system, there is eventually at least \(N\) nodes that are aware of being a leader, and these nodes are non-faulty, \(N\) being a number, whose value will be discussed later.

- **Weak agreement**: among all the nodes of the system, all the non-faulty nodes are eventually aware of the existence of at least \(N\) leaders, and those leaders are not require to be the same from one node to another.

The condition of existence implies that any failure of leader is not necessarily leading to a leader election, as long as there is another leader in the group and all the nodes are aware of its existence.

To summarize the different tools that we have to run such an election protocol:

- Every node has the knowledge of a partial view of the network, and is able to know which nodes are likely to be non-faulty in this view thanks to the state-tagging brought by SWIM and extended in 4.1.3

- The failure of nodes can be eventually detected thanks to the membership discovery.

- A high-probability reliable broadcast provides a fast messaging to all the nodes of the system.

Since the failure of nodes can be detected, a leader, being a node of the system, can also be monitored. Its failure will be the event potentially leading to a leader election. Thanks to the broadcast service, a node elected as a leader has the ability of broadcasting its victory to the system, and can in that way inform all the nodes that it is a new leader. With the knowledge of a part of the system, every node can carefully exchange messages with specifically chosen nodes to lead to the election of a new leader. If we keep a global view on the behavior of the leaders, we can identify two main scenarios where an election is required:

- The number of leaders in the system drops below \(N\) leaders, because of normal termination, node failure, or network failure. In a real case scenario, this case will be slightly different, as we will discuss later, but the idea remains the same. The number of leaders must be rebalanced, a new leader must be elected to compensate for the loss.

- All the leaders are dead, and an election should rebalance the number of leader again.

While the second case seems an extension of the first one, the difference will be important in our leader election protocol, and those two cases will result in two different elections protocol. The first protocol is in charge of regulating the number of leaders, and will take place among the leaders. The leaders, using the failure detector provided by the gossip membership discovery, will be eventually aware of the death or one of them. In that case, another leader should be elected.
A leader alone cannot decide to elect a new node, without agreeing on a kind of consensus. This is why the leaders have to decide as a group, or at least one of them have to decide for the group. Since we are in an asynchronous network, it is not possible to use simple timestamp to order the messages coming from different nodes and then find a trivial solution to make sure that only the first node who decide to be a leader is a leader. The notion of time is not relevant.

In order to describe the protocol of the election, we need to describe the system with relevant details. We will use node identification to sort the nodes. We will see later the details, but every node joining the system possesses a unique ID that can be ordered. Every node possesses a partial view of the network, and knows about the active leaders in the group. Let’s assume that a leader Lx detects that another leader Ly has failed, the following scenario occurs:

- The leader Lx compares its ID with the one of the remaining leaders, excluding the dead leader.
- If the ID of the Leader Lx is higher than the others leaders’ IDs, then the leader Lx starts asking a random normal node – defined as Nc – among its partial view to become a leader. If no answers are received or if the node Nc declines the leadership, the leader Lx repeats the process until a node accepts or until the number of leaders is balanced. If the node accepts the leadership, it is elected, and it will broadcast its victory itself to the nodes in the system.
- If the leader Lx has not the highest ID among the leaders, it will ask the highest leader to run the election process described on the previous point, since this leader will be the highest ID leader. Note that in the message, the ID of the dead node is sent with the request so that the receiving leader can exclude the dead leader from the leader list when comparing the IDs.

A node must have a high probability of accepting a leadership by definition, since the acceptation process is decided when implementing the algorithm. In that way, only network failures could prevent a leader to receive the answer of a node. In case the node accepts the leadership, but the Ack message is lost, the correct number of leaders in the system will be reached again and the election process can stop. With the partial view being continuously updated by the membership discovery protocol, the set of nodes to ask will change and the probability that several nodes refuse the leadership in a row is low. Since we assume that the partial views are uniformly distributed and are update at each cycle, the probability for a node to refuse in a partial view is the same than the probability of a node to refuse in the entire system. Since this probability is low by definition, asking several nodes refusing in a row is unlikely to happen. The probability of having to ask another node is the sum between the probability of the node to refuse and the probability of a network or node failure. We then have

\[ P = P(r) + 2 \times P(md) + P(nc) + P(l > t) \]

with \( P(r) \) the probability of a node to refuse the leadership, \( P(md) \) the probability of a message drop, \( P(nc) \) the probability of the node to be faulty and not yet detected, and \( P(l > t) \) the probability of the latency of the exchange to be greater than the timeout leading to picking another node. We finally see that the process has a high probability to stop after a low number of rounds, making the election process fast. Since a low number of leaders are in the system at the same time, usually close to \( N \), the number of incoming messages to the highest ID leader is low and will not overload the network or the node itself. For the election win broadcast, the load on the network has been discussed in part 4.2.

Now we have seen the behavior for a normal case election scenario. But in this scenario, leaders
are electing the new leader. What happens when multiple crashes occur and no leaders are remaining to run this protocol? Another election protocol had to be described. Unlikely to happen, this election protocol is a security to always ensure that a leader can be elected. Let’s call $Nd$ the node detecting the death of the last leader. The goal with this election is then to elect the highest node as a leader as fast as possible. Since every node has only a partial view of the system members, nothing guarantees that $Nd$ knows about the highest id node.

- If $Nd$ has not the highest ID among its partial view, it will then simply ask its highest id neighbor to run the recovery election.
- If $Nd$ is the highest node in the partial view, it will elect itself as a leader.
- The process repeats to the node receiving the recovery election request.

There are several points that need to be discuss in the election protocol. First, it is possible that a node has the highest id in its partial view but yet is not the highest id node in the system. This could result in several leaders elected at the same time. It is not a problem since our protocol globally provides multiple leader election. The second point is that the highest ID node will receive messages from all the nodes detecting the leaders number dropping to 0 and having the highest ID node in their partial view. If this number is high, it could flood the node. The way the membership protocol is built, in cycle, is forcing the death of a node to propagate every cycle. Since this propagation is progressive, only few nodes will be aware of the death of all the leader in a first time, and the fact that the leader election is not built with cycles will help when it comes to dealing with the flooding problem. The election is likely to occur in less than a cycle, and the broadcast of the winner will be fast enough to prevent too many nodes to launch a recovery election.

To ensure leader knowledge, one could think of a way of sharing the leaders information through the nodes, like the gossip dissemination, but that is a problem, since we need to have a fast detection of a leader death, we should reproduce the same protocol than the gossip protocol to be sure that a leader that is added from another node’s view is not already dead. This would not be cost efficient, nor time efficient, since it takes time to validate death in this protocol.

The question we need to solve is now: how can we identify elections, and then make sure that a node does not run an election if a winner has already been chosen? We introduce an election number, global in the system, that tags every election that is run. This election number, initially initialized at 0, will be incremented only in some specific cases, and will be used for a node to know if he should drop any election request or process it. The election number can be incremented only by a node winning an election. The node will then broadcast its victory with the incremented number included in the broadcast. That way, a node receiving a win from another node will take the win into account only if the election number is greater than the one the node has saved locally. Otherwise, it would mean that the result was an older election and might not be accurate anymore. This is also thanks to this number that a leader receiving an election request from another leader will be able to know if he should run the election protocol or not. The global election number represents the number of the election that has to be done. That means that every node requesting an election will send with its request the election number it has. A node receiving the request can check if the current election should be this number, update its own election number if the incoming election number is greater than the current one and process the election, drop the request if it is lower.

The first node to join the network will be provided with information from the bootstrap node seen in 4.1, and the initialization of the leader service will start by trying to get the list of existing leaders in the system. This will be done by asking another node of the network for the list until one answers. Finally, we will add a process that will periodically check for the number of leaders
in the system using the leader list, and decide to run election if the number of leaders is below the number $N$ defined in the conditions of our leader election. This is to ensure that any edge case that could lead to a lower number of leader would not jeopardize the Existence condition.

### 4.3.2 Evaluation of the solution

In order to evaluate our leader election, we will proceed in several measurements to see if the leader election protocol meets the requirements of our global solution: low load on the network, scalability, robust to node and network failures, real time. We will measure the number of leaders in the group, in normal conditions, with no nodes crashing, but with different parameters of message drop, with a growing number of nodes in the system. We use for the membership discovery protocol: 1.5 seconds to wait for a PONG message, 3 seconds to wait for a PONG message via the bridge nodes, and 25 seconds for a node suspected to be declared as dead. The broadcast protocol will use a fanout of 10 and a number of rounds of 10. We also ensure 15% of leaders in the group of nodes, with a minimum of 2 leaders.

![Figure 5: Number of leaders with a growing number of nodes and different network settings](image)

We can see in Figure 5 that the message drop rate is not affecting the number of leaders in the group. It might seems unrelated, but with high message drop we have false positives and leaders can be falsely detected as dead. In case this happens, we see that the number of leaders is still ensured. Moreover, we can see that crashing all the leaders everytime we add 10 nodes (and reconnect them after) does a effect the global number of leaders, but it is easily explained by the the multiple elections leading to leaders being elected. First, when all leaders are dead, the recovery election can elect more than one leader. Then, the minimal number of leaders required
by the system parameters ensures that others leaders are elected. We see that when 50 nodes were in the network, the number of leaders was too high compared to the minimal required, this is why after the crash of all the leaders and introduction of 10 more nodes, we observe that a fewer number of leaders are elected. Globally, we see that this election solution is able to elect and maintain a number of leaders above a certain threshold defined when building the solution.

4.4 Section Conclusion

With this solution, we have an election that meets the requirement of our global solution, scaling well since it is built on two scaling services, the membership discovery protocol and the broadcast. Robustness is also secured with this construction, while adding another level on the leader election protocol. We have built a solution able to maintain leaders elected, reacting to leaders failure in order to ensure that our bridge nodes, playing an essential part in Monito as seen in 3. This leader election along with the membership and broadcast services set the base to Monito, by making sure that all the requirements are satisfied. We have a growing number of connections and data on the network not increasing more than linearly with the number of nodes in the network. We keep close to real time in all those services. We can continue to build our final solution based on the protocols created so far.
5 The use of leaders to ensure data flow

It is now possible to use the leaders provided by the leader election service, and finally give them a special role as we wanted in the previous section 3. We now have an architecture that is clearly presented in 6.

Figure 6: Overview of the monitoring architecture
There is the system of nodes, linked through a message queue to the main server with which the user can interact to access any data coming from the node. We see that the group of node is composed of two kinds of nodes, normal nodes and leaders, playing the role of special nodes in the system. But if we have all the tools to create this structure, we now need to know how to deal with the data coming from the nodes. The objectives of this section are to find a way of correctly ensuring a flow of data from one node to the message queue of our solution. There are several requirements for the data flow solution:

- Every data from a node should reach the message queue at least once, explicitly or implicitly through an aggregation.
- The data should reach the message queue in real-time, real-time being defined here by an amount of time inferior to a minute.
- Aggregation should be provided by the solution to reduce the data throughput if needed.
- Filtering should be provided by the solution for the same reasons as the aggregation.
- We keep the solution requirements: scalability, robustness, avoid adding external components, low resource usage.

The global idea is quite straightforward, and close to the Datadog flow: the nodes send the data to the leaders which buffer the data, aggregate it if needed and send it to the message queue. As easy as it seems, providing robustness to node and network failures while keeping the solution real-time is a problem that will be solved by a specific protocol.

5.1 The leader as a buffer

Every node elected as a leader, will have the capacity to receive data from others nodes and send it to the message queue, by initiating a connection to the message queue. To avoid too much transmissions with the message queue and keep the scalability easy, it is better to increase the size of the message and lower the number of messages exchanged. Since we use RabbitMQ in our solution, the performances of the rabbit message queue [14] clearly show, on the “sending rate message sizes” graph that the bigger the messages are, the higher is the rate of byte per second processed by RabbitMQ and the lower is the number of messages sent per second. That means that less messages but with a bigger size are more efficient than more messages of a small size. In order to increase the size of the messages sent to the message queue, while maximizing the volume of data sent, buffering the data at the leaders before sending a packet of data to the message queue is an interesting idea. The buffer can easily run an aggregation protocol on the data, and then we optimize the solution to fit our requirements. In order to equally share the resources cost of a leader, the data from the nodes of the system has to be equally shared among the leaders. For that, we ensure that a node sends its data to one of the leader, picked randomly among its list of leaders. This random selection ensures that with a high number of nodes compared to the number of leaders, all the leaders will receive approximately the same load of data through time.

To reduce the amount of data on the network at a given time, and reduce the incoming messages to the leaders, we add a bufferization at any node level. When data is produced by a node, this node will buffer the data before sending it to one of the leaders. It is then possible to optimize the message sent by removing redundant information such as the source of the data, since all the data in a packet will come from the same node. In that way the load of the network is reduced.

\(^6\)“low” being defined here by “not impacting the services originally run by the nodes”
In both a node or a leader, the buffer can be cleared and the data sent when it is full, or after a timeout. In Monito, the buffers are cleared the first time one of these events occurs. We can easily deal with filtering in the buffer, but we chose in Monito to filter the data when it is read, keeping in memory the last value read for each type of data, and applying a custom filter to drop or process the new data read, comparing it to the last value read. Finally, we need to solve the main problem of this protocol: how to ensure that the data is reaching the message queue at least once? For that, we use a specific protocol. Let’s consider that a node Na has a packet of data <Pdata> to send.

- The node Na picks randomly a leader among its leader list, excluding any leaders in a set of unavailable leader and itself 7.
- The node Na adds the leader to the set of unavailable leaders
- The node then wait for the leader to send an Ack message showing the success of the transmission to the message queue.
- If the node receives a positive answer, the leader is removed from the list of the unavailable leaders.
- If the node receives a negative answer or a does not receive any answer after a given amount of time, another leader is selected as in step one, and the process is repeated.
- If there are no more available leaders, the node waits for a given amount of time before retrying. If no leaders are available, the node checks is leadership, if it is a leader he sends the packet to its own data handling service. If not, the node elects itself as a leader.

On the leader side:

- The leader receives the data and store it in a buffer, adding the sender to a set of senders.
- When the buffer is full or after a given amount of time, the leader aggregates all the data in the buffer, each data type having its own rules of aggregation, that can be to not aggregate anything.
- The aggregated data is split in packets that will be sent to the message queue
- The leader is now in sending mode, it refuses any incoming data before terminating sending all the packets to the message queues
- Thanks to RabbitMQ it is possible to ensure that the data sent to the message queue reaches it. If one of the packets fails to be sent, the process tries to send the remaining packets.
- If all the packets have successfully been transmitted, the leader acknowledge all the nodes in the senders set positively, negatively if one of the packet failed to be sent.

This ensures that the data reaches at least once the message queue. The data may be duplicated in case of a failure with the transmission to the message queue, or in case the acknowledging message is dropped or too much delayed. But this is not an issue regarding the Monito requirements. The global intercommunication protocol is described in the following graph:

---

7This set is updated in the following steps of the protocol
5.2 Evaluation

We will first evaluate the redundancy of the data in our solution, testing it on a network simulation Mininet of 40 nodes and a latency up to 400ms. The message drop will have different measures. The settings for the protocol will be:

- A bufferization time at the leader of 8 seconds and a buffer size of 16KB.
- A bufferization time at a normal node of 8 seconds and a buffer size of 8KB.
- Around 200B of data produced every second at a node, and data will be produced during 60s.
- A timeout of 10 seconds on the unavailability of a leader, meaning that a normal node will wait 10 seconds before considering that the packet failed to reach the message queue.
This choice of settings leads to a configuration where the buffer is kept a maximum amount of time before being cleared since the rate of data sent is very low. It is the case that is one of the most relevant since the timeout for receiving an Ack from a leader will be just slightly more than the maximum time data is kept at the leader buffer, leaving enough time for the exchange of messages to happen with the maximum of latency. The redundancy rate is calculated as the average for all nodes of the data that is received more than once at the main server side. That means that the measures have been taken for each node, and then the average of redundancy has been computed.

![Figure 8: Percentage of data redundancy with an increasing rate of message drop on the network](image)

We see that the redundancy of the data follows the message drop rate, since messages of acknowledgment can be lost, forcing the node to resend information to another leader. We might think that this percentage should be twice more than the drop percentage, since two messages can be lost (the incoming data and the Ack from the leader) when going through the network, with message drop twice more likely to happen if we consider this double message exchange (7% in one way, 7% on the other way). But if the message is dropped in the way in, then no data reaches the main server. In a real situation on a regular network of good quality, the redundancy of the data will not be important, and the network will not be overloaded with this redundant data.

We will then evaluate the behavior of the leaders with a lot of pressure, starting with only one leader in the group and see how the number of leaders evolve with the time. We reduce the buffer size at a node to 500B, and at a leader to 3500B, and increase the rate of data to around
600B per second. We keep a message drop of 2% and a latency of 400ms.

We observe, for the 20 first nodes, that the number of leaders is way higher than the normal number of leaders that should have been elected looking at 5. This is explained by the difference between the size of the buffer at a leader and the size of a buffer at a node compared to the amount of data produced. Indeed, every second, a node has to send its data that is not fitting in the buffer. A leader receives this amount of data from several nodes, with an average of $\frac{N_n}{N_l}$ this amount of data per second, $N_n$ being the number of nodes in the system and $N_l$ the number of leaders. During one second, the leader does not receive enough data to send it straightaway to the message queue. He will then buffer it. However, the nodes are still in need to send data to leader every second. A leader not being available before it Acks the node back once the data has been sent to the message queue, the node will have no choice but to send the data to another leader. Once all the leaders are unavailable, the node elects itself as a leader to handle sending the data to the message queue. However, since every 8 seconds a leader has to send the data, we can ensure that a leader stays unavailable no more than 9 seconds (with latency). So after 9 leaders are in the group, it is normal that a node will finally find one of the leader available, reducing the need for leaders. Moreover, with the increasing number of nodes, more data is sent to the leaders, meaning that the buffer will be full before the sending timeout happens, making it available again. We see that it is important to carefully chose the parameters for the buffer size, compared to the global volume of data that has to be sent through the network.

Figure 9: Number of leaders with an increasing number of nodes in the system, and stressful data flow conditions
5.3 Conclusion

Thanks to the leader election service, we are able to create a protocol for sending data to the main server through the message queues, ensuring that the data will reach the message queue at least once even with bad network conditions. This solution using nodes as bridges and temporary buffers scales well, by allowing aggregation at every leader node, and filtering data produced at every node. A bigger system only requires more leaders, and with the leader election protocol satisfying or initial conditions, we are able to provide another layer of robustness to node crashes and network failures. With the buffer being temporary, we allow our solution to produce close to real-time results, without adding any external component for the data flow protocol than the one added for the other services. The number of connections is still not growing more than linearly with the number of nodes, and the load at each nodes is divided even for a leader, by having nodes picking a random leader at every cycle of the data process. With data being able to be sent from nodes to the main server, we now need a way of producing this data, and be able to control the data produced at every node, to have a relevant monitoring service. This is done by applying a test deployment solution on top of the current architecture, and will be discuss in the next section.

6 The data flow, base of a test deployment protocol

This section will be dedicated to the creation and evaluation of a test deployment service on top of the Monito architecture. In the previous sections, we have created and evaluated an architecture and a set of protocols ensuring the transmission of data from a set of node to a server, in an environment with potential network failures and node crashes. The solution is robust and scalable, while running almost exclusively on the data producers node and not impacting their original services with high resource consuming. We will now explain how to use these protocol and architecture to create a reverse data flow, ensuring that data produced by the main server –tests–, will reach all the nodes in the system.

6.1 Objectives

For this test deployment solution, we keep the same requirements than the global requirements of Monito. We want the solution to scale as well as possible, with a priority on robustness to node crashes and network failure. As for the previous parts, avoiding external components is part of the challenge of the solution. The goal of this solution is to be able to realize two actions:

- send a test to all the nodes in the system.
- send a test to a specific node in the system.

While the latter action seems to be included in the first one, since all the nodes receiving a test means that the targeted node will receive the test as well, but we have to avoid to overload the network or the node with unrelevant actions or data storage. This is why we can’t afford to have all the nodes in the system receiving a test if only one should receive it, the main reason being that if we send a lot of tests to different nodes, the network could be flooded. We will see how to solve the issue by adapting the test diffusion to a global or unique destination.

6.2 A test deployment without external component

Before starting this section, we will introduce two terms:
• The “deployment cycle” as being the theoretical cycle starting with the fact of pushing a list of tests to the message queue and ending with the test being deployed on all the nodes of the system. A cycle is considered as terminated once all the nodes able to receive the tests—taking into account network and node failures—have received the tests.

• The “cycle descriptor” that is the content of the message pushed to the message queue, containing for instance the list of tests.

With a broadcast service and a partial view of the network, we have strong tools to disseminate a test among one or several nodes. The challenge is to decide of an entry point, as the main server deploying the test is out of the system. The way we define our test deployment as a “reversed data flow” is an interesting point to take into consideration. We have an architecture with special nodes, the leaders, that can be used as entry point, since they already communicate to the main server through message queues. This time, we need to add a message queue that will push messages, in our case: the tests, to the leaders. With the fanout option of RabbitMQ, a message queue can push a message to all the listeners listening to a special tag [15]. This great feature will enable the leaders to listen to tests, and pull the tests from the new message queue. As for the first data flow, we could have all the nodes listening to the message queue, but ensuring scalability is a key requirement in our solution, and having all the nodes connected to the message queue would overload the message queue way faster than having only some leaders connected. This is why we chose to have only those same leaders playing a double role, pushing data and pulling tests.

Now, there is an entry point from the main server to the system, but the problem remains to distribute the test to one or all the nodes by limiting the load on the network and delivering the test with a high probability of success. We will first limit our study to the case where all the nodes should receive the test. Broadcasting the message to all the nodes should be avoided since our broadcast solution produces a non-negligible rate of redundancy. The solution that can avoid to broadcast the test is to only broadcast a header, or a test availability message, and then distribute the tests to the nodes in a less redundant way. To avoid network failure problems and ensure that the test are correctly delivered to all the nodes, a TCP connection will be set up between the test provider and the test receiver. In our case, the test providers are the leaders themselves, receiving the tests from the message queue. The receivers are all the normal nodes of the system. A test or several tests are created in the main server and pushed to the message queue. The leaders receive the tests. A broadcast message is sent to the system through our broadcast component, making sure that all the nodes receive the message with a high probability. The content of the message, as discussed before, is not the tests files themselves but a description of the tests files, used to identify them. We choose to use the test name as the identifier in the message. The next step consists in getting the test from a node. What the system knows is the identity of the current leader.

By definition, all the leaders that received the test – and did not crash during the short time period needed for a node to receive the broadcasted message – have stored the tests, since they are actually part of the system, they are therefore receivers as well and need to run the tests. Any node contacting a leader could then be able to download a test. In the first sections, we have decided that data was sent to a leader with a random selection. Among all the leaders, data was sent from a normal node to a randomly picked leader. The same process is going to be used for the leader providing the test. A normal node chooses randomly a leader and attempts a TCP connection to a system-wide defined port. If the leader accepts this connection, the downloading protocol begins, if not, another leader is chosen. Once the connection is accepted by both sides, the receiver sends the list of test-files that it still needs to acquire. The leader checks which tests are available on its side, and begin to send all the tests back to the node. With TCP, we ensure
reliability over the UDP protocols used in the others services and scenarios. If the leader does not possess all the tests needed by the receiver, the latter will contact another leader later on, and repeat the same process until its test-files list is completely downloaded. The entire process is summarized in 10.

Figure 10: Test deployment protocol

The consequences of this protocol are directly impacting the way of distributing the tests among the nodes. An important part of the setup of such a protocol remains in the way of handling the tests once downloaded by a node. That is up to the programmer to decide which policy to apply regarding the time of storage, but some points need to be discussed concerning some side effects of this protocol:

- A test distributed with the same name than another test is not gonna be deployed if the previous test with the same name is still stored on the node. Actually, the problem comes from the leader node, which needs to know what test to distribute when asked for a test by ID. If it seems trivial to just consider the tests that have been downloaded from the message queue on the same “deployment cycle”, it is not since with message reordering and multiple test cycles launched in a row, it is not possible to clearly identify which tests are supposed to come from which deployment cycle. Another solution could be introduced, by prohibiting two tests to have the same name, but it requires to keep track of the test id sent at the main server level.

- The nature of the network and the system is leading to network failures and node crashes.

37
or disconnections. That is why we need a robust solution, and in that case we may run into trouble if all the leaders crash right after a test had been pulled from the message queue, or between the test-availability broadcasted message and the test exchange through the TCP connection. A solution will be discussed in the description of a second deployment protocol.

- One notion that was not discussed before but remains important is the security of our protocols. In the case of the test deployment, having an attacker sending a false test to a node could result in the node executing an attacker’s code with potential damages to the system or the node itself. Signing the test with a certificate could be a solution, but we will not go into details since the second version of the deployment protocol brings a more secure way of deploying the tests.

With randomly distributed connections, the protocol provides a good way to avoid overload on specific nodes of the network. Avoiding the broadcast of the entire test-file, the protocols offer an interesting way of scaling to a high number of nodes in the system, as long as the leader service provides enough leaders. Broadcasting the test-availability message can be done whether from every leader, leading to a big amount of redundant data on the network, or only by a subset of leaders using the ID orders (highest ID leader broadcasts only, for instance, as it will be chosen in the solution). This comes with a cost: less reliable because of leader crashes. We will not measure the impact of the number of leaders broadcasting on the network and let the programmer makes his own set of tests depending on his requirements.

As expected and shown by 11, the number of connections per test deployment cycle in the system is increasing linearly and at the same rate. That is due to the nodes that still need to connect to a leader even after the tests descriptions have been broadcasted. While it stays as a linear growth, having the number of connections reduced is important to fit our global solution. However, this number of connections are mainly impacting the network, but with the leader all participating in the process, we avoid bottlenecks and still have a linear growth of connections at every leader but divided by the number of leaders in the group. If we use the graph of the leader election section, concerning the number of leaders elected function of the number of nodes in the system, we have an interesting graph showing that the number of connections per node stays approximately constant. When we introduce leaders failures however, the number of connections is growing with a multiplicative factor as seen in 11, since less leaders are able to distribute the tests. With our requirements being to be able to handle node failures, we can conclude that this solution is not optimal for such a robustness policy. However, the TCP protocol provides good robustness to network failures, as we can see that message drop rate does not influence the protocol in 11. In our experimentation, we chose to ensure a high probability on the broadcast success to focus on the test deployment part of the protocol.
To end this evaluation of our protocol, we can see that the test deployment is almost real time, and seems independent from the number of nodes in the network, as shown in figure 12. This is not really the case since the broadcast takes a small amount of time more to reach all the nodes when the number of these nodes increases. But we stay close to real time, and the increase is insignificant after a certain amount of nodes (around 10) are already in the network.

The measurement has been done in our mininet network simulation with a 0% message drop rate and 70ms latency. By definition, a deployment cycle is measure by the time difference between:

- The timestamp of the download of the last test of the test list on the latest node to receive all the tests in the system
- The timestamp of the moment when the test list has been pushed to the message queue.
No more components have been added to our solution, fitting the last requirement of the global solution. But with the problems discussed about security and leader death, another solution could be used, using this time an external component: a Content-Delivery Network.

### 6.3 A more secure CDN-based protocol

It is becoming easier for companies to afford cloud based solutions, and third party servers, with specificity like robustness, resources or availability being easy to setup (like AWS [2] and Google Cloud [7]). This is why this paper introduces a test-deployment protocol, more secured than the first one, based on the use of a CDN. The global idea of the delivery system remains the same. The part that is different is the way the content of the test is distributed to the nodes. Instead of pushing the entire tests to the message queues, only the descriptors of the tests are pushed. As in the first test-deployment protocol, we use again the name of the test-file as the identification number of a test. We will add to the cycle descriptor an ID, based on a counter, from 0 for the first test cycle and incremented every cycle. Most important, the cycle descriptor now contains not only a test ID for each test, but also a Hash of the test-file. This time, every leader will store a given number of the last test cycle descriptors, that will be decided by the programmer. The new protocol consists in broadcasting the cycle descriptor to the nodes, as before, but the behavior of the receivers and leaders is completely different. The leaders and normal nodes, when receiving the test descriptors, from the message queues and broadcast service respectively,
will check if a test with a same test identifier is already stored. If yes, the node hashes the local file and compares it to the hash in the test descriptor. If the hashes match, nothing is done. If the hashes are different, or if the file does not exist locally, the node will download the test from the CDN, using the test name and some configuration parameters to build the CDN download address. Building this address is not in the scope of this paper. Once the test file has been downloaded from the CDN, another check comparing the hash of the downloaded test and the test descriptor hash is done to ensure that the correct test was downloaded. The entire process is summarized in 13.

![Diagram of test deployment with a CDN](image)

Figure 13: Test deployment with a CDN

This process allows an override of a test file that is way safer than our first protocol since the CDN will be the only component needing to have the right version of the test file, and all the nodes will not have to refer to test files downloaded from leaders. Along with security, this protocol removes the need of TCP connections to the leaders, but leads to a lot of connections happening at the same time to download the test file from the CDN. One major issue with the first protocol was to be able to handle leaders failure, especially the failure of leaders broadcasting the test-message. One problematic aspect is that we don’t have an active polling of the message queue but a passive one (the queue contains a message that is pushed to the leaders, and the leaders can’t access a message already pushed). To solve this solution, we need to look at the mechanism keeping the system up to date with node deaths: the membership discovery protocol.

Thanks to that protocol, we know that eventually, a dead node will be declared as dead. As we saw with SWIM, the time to be aware of a node’s death is tightly linked to the timeout changing a suspected node to a dead node. This timeout being a configuration parameter of the solution,
it is easy to estimate the time it takes for a node to be detected as dead, by summing the timeout for a node not answering to a ping to be suspected (after not answering to a pinged by the bridge nodes), and the one from suspicion to death. Once we have this value, we use it to push the test descriptor a second time in the message queue after an amount of time corresponding to the value found is spent. We do not eliminate the risk of a test not being sent, but yet reinforce the probability for the nodes to receive the test eventually. We define the time to detect a leader death to be the time between the leader crash and the first leader being aware of the event and launching a new leader election. We see in 14 that the approximate time in our solution to detect the death of a leader is independent from the number of nodes in the system. The measurement has been done with 4% of message drop and 150ms of latency. We can conclude that pushing the tests to the message queue a second time 36-40s after the first time, leaving time for a new leader to be elected will greatly increase the probability for the nodes to receive the messages.

![Figure 14: Time to detect a leader’s death with a growing number of nodes in the system](image)

With the probabilistic broadcast solution, there is still a risk that the message would not be broadcasted to all the nodes. One solution to improve it on this layer is to have the normal nodes keeping track of the last received test, and ask a leader for a test cycle descriptor if the node receives a test before receiving the one with the lower ID. This solution also ensure that even if the test descriptors broadcasting leaders die, a test would eventually be delivered as long as one of the leader which pulled the test is still alive and still store the missing test descriptor. Keeping track of the last tests cycles also makes sure that the test service does not accept the tests several times in case of the test descriptor being sent by different leaders. (The broadcast service is only preventing redundancy from the same broadcast source.)

We can now measure the percentage of nodes receiving a test cycle with a growing number of node and several network and nodes failure.
Figure 15: Percentage of nodes receiving tests after a deployment cycle with a growing number of nodes
Figure 16: Percentage of nodes receiving tests after failures events

In 16, events 1-2 are failures of random leader nodes, making sure that 25% of the leaders remain. At event 3, all the leaders are crashed right after the pulling process from the message queue. Event 4 is as event 3, plus another crash of the new leaders elected after the pull of the second test message in the message queue. Event 5-6-7 are normal test deployments.

We see that all the nodes receive the test even with leader and network failures thanks to the robustness of the broadcast and its test deployment improvement, and thanks to the second test deployment system. However, if we crash all the broadcasting leaders a second time during the second test push, the test is not distributed until another test is pushed to the queue. This is happening because of the broadcast-improvement mechanism: the cycle missed is still stored at the leaders that were not broadcasting and stayed alive. So when a new test reaches a node, this node is aware that the previous test is missing, he can ask leaders until he gets the test descriptor, and give up after a while if no leader seem to store this descriptor anymore.

Before concluding on this test deployment service, we still need to find a solution for a one node test deployment. The same process will be applied until the broadcast part. Special test ID are used in that case to not interfere with the ID of the system-wide deployed tests. The idea is that the leaders, pulling the tests descriptor, will proceed to a probability calculation: If the number of leaders in the group is enough for the message to have a probability $P$, close to 1, decided by the programmer, to reach the targeted node, then all the leaders send a test description to the targeted node. If not, they send the message as well but ask for the node to answer back, repeating the process until the node acknowledges the reception of the test descriptor. The probability for the node to receive the message is $(1 - Pn)$, where $Pn$ is the probability that the node does not receive the message, so that the message has been dropped
or the leader has crashed. It is calculated that way:

\[ P = [(1-c) \times (1-d)]^N \]

where \( N \) is the number of senders, \( c \) is the probability for a node to crash and \( d \) the probability for a message to be dropped.

In that way we ensure with a high probability that the node will receive the message, without overloading the network with irrelevant acknowledgment in case a lot of leaders are in the system. We will not test this solution as the settings are quite straightforward to decide and cannot produce unexpected or interesting results.

### 6.4 Conclusion

Using our broadcast service as test broadcasting and the leaders as entry points in the system, Monito is now able to provide a reliant test deployment service, targeting a special node or all nodes of the system. By using the architecture built for the other services, this test-deployment service benefits from all the advantages discussed on the previous sections: scalability, low load on the nodes and network, robustness to handle failures. Two solutions have been presented, one lacking a security layer and potentially limiting the size of the tests in order to not overload the network, and the second more secure, and keeping a small size of messages sent over the network to deploy the tests, but with the cost of adding an external service, a Content Delivery Network (CDN). This CDN could be a bottleneck in the solution, but its overloading may just result in tests downloaded in a longer time with a huge number of nodes in the system, while ensuring that the test identifier itself is deployed on the nodes without being limited the same way by the size of the network. The limits of the broadcast service built in Monito are compensated partly by the test deployment protocol, ensuring a better message delivery reliability. We will see in the next section how we can ensure scalability in a higher way than what was discussed in the previous sections.
7 Scaling the election leader protocol

If it was said that the previous solutions for the services described in the previous parts were scalable, it is true only for a medium number of nodes (100 at least in our experimentation, more if based on the gossip papers). But our initial requirement was to have much more nodes running in the solution, and being able to monitor and test them while meeting the others requirements. We will try to discuss solution to add another layer of scaling using the components that we saw in the previous sections.

7.1 Division in groups, the bootstrap node as a key node

The approach for that last scaling layer is to use a central solution. We have seen that the bootstrap node was the key to ensure the well-behaving of the gossip membership discovery, and this discovery is the base of all the services built in Monito. The advantages of the bootstrap node is that all the nodes joining the system are interacting with it before entering the system. The bootstrap node is the component making possible for a node to join the network of nodes by providing some of the existing nodes in the system. It is the ideal component to use for the purpose of scaling our solution.

Monito runs and scale until a medium number of nodes quite well. The idea is to keep these protocols as they are, since they prove useful, but to introduce a division of the system in order to run them across different group of nodes. What makes a system interacting through Monito protocol is the fact that the nodes are interconnected thanks to the gossip service. This gossip service being initialized with neighbors nodes, it is easy to keep a protocol running over a medium amount of node by controlling which neighbors are provided to a new node joining the network. That is why the solution could be to divide the nodes into groups at the bootstrap step. After a group is full, a new group is created, and a new node joining the network would enter the new group as if it was entering an empty network. The node would not be aware of the others nodes in the others groups, but it is not a problem since the groups would be linked thanks to the leaders of each groups communicating with a common message queue. In the end, the bootstrap node contains a list of partial views, and any incoming message from a node needs to be processed to update the correct partial view. The node communicating to the bootstrap node simply needs to add its group ID in the information sent and the bootstrap node is then able to update the correct partial view. The bootstrap node is still having a passive listening to all the nodes and does not participate in any gossip to avoid irrelevant processing load. One might point out that we are turning from a distributed solution to a central one, but a bootstrap node crashing would not a ect the flow of data, since the groups are not using the bootstrap node to transfer data and receive tests. Moreover, thanks to the groups keeping the bootstrap node updated as part of the protocol described in the previous sections, it would be able to restore automatically the network schema once the bootstrap node has recovered.

The summary of the content of a node and a bootstrap node are compared in fig 17 and 18. In our final solution, we chose to have 100 nodes in every group maximum, and 150 nodes deployed on 2 groups, a partial view of 40 nodes, exchanging up to 20 nodes of each state to others nodes. The gossip timers values are the same than in fig 1. The broadcast fanout is 10 with 10 rounds. The buffer size is 16Kb at a leader, and 1.6KB at a node, and a buffer time of 8 and 10 respectively. Since we already have discussed the results of the intermediate services, we will try to see the average of time needed for data to come back to the main server once a test has been sent to the message queue, with 4% message drop and 400ms latency, and a minimal number of leaders at 7%. We deploy several tests, with the CDN protocol, and measure the average time. We obtain an average of 15 seconds. This is due to the fact that only one test is
deployed at a time, not filling the buffers. With the latency and time to process and deploy the tests, we obtained a close to real time value even with message drop, satisfying our requirements. In the process, we targeted one group of nodes only, and received the data from the nodes in the group.

7.2 Avoiding topology problems

One of the main concerns when building such high scale solutions implemented over planetary sized networks is the topology. A lot of devices can prevent our solution to run, and a lot of rules have to be applied to make sure that all the needed communication is possible between each node. The most important problem is the presence of NAT devices on the network. Such devices, by providing Network Address Translation, prevent the nodes to be reachable from outside the NATed network, except for some public addresses. While some solutions exist to enable communication from the external side of the NATed network, they are quite difficult to scale. A KTH project, called SWIM through NAT offers a solution using public nodes (nodes with public IP) as bridges to communicate between the two sides of the NAT. While interesting, this solution only works when public nodes are present in a large percentage in the global system of nodes. That is not necessarily the case for our solution, which must be able to work in any topology configuration. But thanks to the last scaling step presented in the last section, the system is divided in groups, and the requirements for communications only apply to a group, since all the others communications are initiated from the monitored nodes toward the external components. As long as those components are able to be reached, Monito will not encounter any troubles. It seems that dividing into groups is therefore not as easy as it could. We cannot afford to just wait for a group to be full before going to the next one, we need to make sure that all the nodes inside a group are able to communicate to each others through UDP. The ideal way of dealing with this problem is to guess the topology of the network, and then be able to identify all the nodes behind the same NAT device. But with routing and load balancing, being able to discover the topology of the network is a real challenge that has been the topic of many scientific papers. The solutions in those papers are complex and often cannot ensure that the topology discovered is the real one, especially in a planetary sized network. The problem being complex and the solutions as well, Monito is not providing topology discovery. Instead, in order to provide a relevant test deployment service, Monito gives to the user the entitlement to decide of the group composition. It may seem a hard task, but dividing in groups is not only a topology issue, it is a logical one as well. Indeed, deploying tests over multiple nodes usually means that those nodes offer a kind of identical service. Regrouping the nodes could then be done in one hand to be sure that the nodes inside the same group can communicate, and make sure that the groups are created with a logical use case profile. But to get rid of having to deal with group size, the bootstrap node must be able to divide a group in several subgroups that would be of the correct size. This is what is happening at the bootstrap node. Every time a node wants to join a group, the node will in reality join the first subgroup with enough space. If no subgroup are available, a new one is created. The node will receive the identifier of the subgroup and will send it when sharing its partial view with the bootstrap node.
Figure 17: Summary of the relevant information contained in a node
As we can see, the bootstrap node contains a higher abstraction of a node, containing multiple partial views for every subgroup.

8 Future work

In this paper we discussed several problems, with a lack of measurements of the network load and concrete connections number and messages exchanged number. This is due to the fact that taking those measures is quite hard to do, in an asynchronous system, without interfering with the measures. A more detailed study on the load handled by the network needs to be done. Related to that point, we see that the broadcast mechanism can be considered as a risky point of flooding the network. The algorithm could be optimized by reducing the fanout at every cycle of broadcast, that might be a better solution even if the number of cycles required to cover all the network is increasing. The impact of the partial view distribution could be studied as well, comparing theoretical uniform distribution of the view through the partial views, to the real distribution.

Another problem not discussed in the previous section is the behavior that the bootstrap node should have when a node joins the network again. If the node is put in a different subgroup than it was before, its old neighbors could still try to contact it, creating a breach in the separations between subgroups. Indeed, the node will still appear in the old neighbors partial views. If one of this node still considers the node as suspected or alive, it could try to ping it as a normal
behavior. The node will then receive a ping from a node, and will register this node in its partial view, while it should not belong to the same subgroup. This should be avoided by maintaining a list of nodes ID and the corresponding subgroup at the bootstrap or server level. When to erase entries in this list, to make scalability possible, is a problem to be discussed.

Concerning the leader election protocol, a solution should be found to resign a leader. Informing the group of the resigned leader and choosing the leader to resign are complex problems. We consider, in our solution, that a leader that resigns would simply stop his leadership services, resulting on the data flow taking care of identifying the leader’s resignation, thanks to the Ack message sent by the latter when receiving data from nodes. This Ack could announce the resignation, making sure that the node which sent the data will remove the leader from its leader’s list. A solution could be to proceed in the opposite “direction” than the regular election: instead of electing the highest ID node, the lowest id node could be resigned.
9 Conclusion

We have been able, though this paper, to build a monitoring and test deployment solution having challenging requirements. We finally obtained a solution:

- With a number of connections in the system being whether independent of the number of nodes in the system, or growing linearly with it, but always at a lower rate.
- With a load on the network constant or, most of the time, that was growing linearly with the number of nodes in the system, once again with a lower rate thanks to aggregation process and communication optimization.
- Robust to network failures: all the solutions presented in the sections have been proved to work under high message drop and high latency without influencing the global data flow.
- Robust to node failures, disconnection and connections: with the membership discovery service, the system map was able to be updated in a fairly short time.
- With a low number of additional components, that is constant even when the number of nodes in the system increases.
- Close to a real time solution, that we consider as real time in the monitoring and test deployment use-case.
- Where aggregation of the data is not mandatory but can be used easily
- Sensible to the network topology, forcing a manual group division of the system to be able to work, and where topology can influence the scalability (One node per group would for instance represent a normal centralized solution). The optimal topology would be to be able to divide the system in groups of partially connected networks, to fit the node communication requirements needed by the services.

Through all the sections, we have presented all the pieces making the solution work, trying to ensure at each step that all the Monito requirements were respected. We have first decided on an interesting way of dividing the data flow by introducing special nodes, communicating with a central unit—handling and processing the monitored data—through message queues. To avoid adding external components as bridge nodes, we have created a protocol able to elect and maintain those special nodes that we called leaders, over a network having potential failures, and with a system of nodes to monitor that is not stable, nodes entering and leaving the network. This protocol, built on a highly scalable gossip membership discovery protocol, makes possible the implementation of a probabilistically reliable broadcast inside of the system. While making sure that those components were meeting the Monito requirements, we started building data flow protocols to gather data from the nodes and deploy test on the system or targeted nodes. We finally added a layer of scaling by bringing a solution to divide the system into groups and subgroups of nodes, taking into account manually the network topology. The final solution, Monito, is a solid way of bringing test-deployment and monitoring together in a scalable and cost-limited architecture, but could benefit from a deeper study on network load and connections complexity.
References


