ISSN 0103-9741

Monografias em Ciência da Computação

nº 06/2023

Design and Implementation of the ContextNet Kafka Core Middleware

Camila Antonaccio Wanous

Markus Endler

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 06/2023 ISSN: 0103-9741
Editor: Prof. Editor Principal August, 2023

Design and Implementation of the ContextNet Kafka Core Middleware

Camila Antonaccio Wanous, Markus Endler

cantonaccio13@gmail.com, endler@inf.puc-rio.br,

,

Abstract. ContextNet is a communication middleware based on the OMG DDS standard, which supports real-time monitoring, unicast, groupcast and broad- cast for thousands of mobile nodes. This work aims to replace the DDS (Data Distribution Service) infrastructure, used in the communication of the Con- textNet Core nodes, with the Kafka platform; simplify the construction of applications that use ContextNet middleware; and add new features. Kafka is a distributed streaming platform capable of subscribing, publishing, storing and processing in real time streams. Kafka will allow applications built on ContextNet to be paralelizable.

Keywords: Cloud-Mobile Communication, Middleware system, Kafka, Scalability, Internet of Mobile Things

Resumo. ContextNet é um middleware de comunicação baseado no padrão OMG DDS, que suporta monitoramento em tempo real, unicast, groupcast e trans- missão para milhares de nós móveis. Este trabalho visa substituir a infra- estrutura DDS (Data Distribution Service), utilizada na comunicação dos nós Core da ContextNet, pela plataforma Kafka; simplificar a construção de aplica- tivos que utilizam o middleware ContextNet; e adicionar novos recursos. Kafka é uma plataforma de streaming capaz de subscrever, publicar, armazenar e pro- cessar em fluxos em tempo real. Kafka permitirá que as aplicações construídas na ContextNet sejam paralelizáveis.

Palavras-chave: Comunicação núvem-mobiles, Sistema de Middleware, Kafka, Escalabiliadde, Internet das Coisas Móveis

In charge of publications:
PUC-Rio Departamento de Informática - Publicações
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: publicar@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

This work is about the reimplementation of ContextNet [1] by replacing the communication middleware standard that it used it with a new technology, Kafka; adding functionalities; maintaining its architectural principles and its main purpose - to support mobile IoT applications.

The original ContextNet was a communication *middleware* based on the *Data Distribution Service* standard of the *Object Management Group* (*OMG DDS*) [2, 3, 4], which supports real-time, unicast, groupcast and broadcast monitoring for thousands of mobile nodes. ContextNet elements can be divided into static and mobile. The former are part of ContextNet Core[5] and operate using the *OMG DDS* infrastructure, which this work has replaced with the Kafka platform.

ContextNet [6] was developed to support the implementation of IoT applications that enable both real-time tracking and efficient means of interaction between thousands of mobile devices, sensors and actuators connected to them. Furthermore, in many of these applications, the set of participating mobile nodes varies constantly, as they can join and leave the system at any time and at any point in the network, either due to application-specific circumstances or due to intermittent wireless connectivity. Such applications therefore require a scalable, reliable and near-instantaneous communication infrastructure and the dissemination of context throughout the network. ContextNet in both versions provides this infrastructure.

Kafka [7] is a distributed *streaming* platform capable of accepting subscriptions to, publishing in, storing and processing *streams* in real time. This replacement was accompanied by the addition of new functionalities to the *middleware* described in the following; by the adoption of the *Publish/Subscriber* protocol for communication between Core nodes; and by the addition of parameterization possibilities for users.

Through this redesign of ContextNet the ease of building IoT solutions using the middleware is combined with the possibility of parallelizing static components and thus improving the performance of ContextNet microservices;

The original ContextNet[1] supports a significant number of mobile connections per gateway (up to 12,000 nodes connected per gateway have been successfully tested) and therefore satisfies the current needs of its users in terms of the number of connections. However, the original version does not allow application nodes to be parallelized and sends all topics over a single channel, forcing all static components to receive all the messages sent, including those whose on topics that do not concern, overloading them and therefore making it impossible to build complex or scalable applications capable of supporting large numbers of mobile connections. The use of Kafka will instead allow the new version of ContextNet to parallelize static components, and therefore allow the development of complex and/or scalable core applications.

The original ContextNet requires extensive and laborious configuration of the DDS environment, which is costly for developers. The relative simplicity of the Kafka implementation compared to the DDS implementation and the use of *Containers* have made it easier for developers to use ContextNet. In terms of infrastructure, one needs only to have *Docker* installed to use ContextNet.

ContextNet's architectural principles have been maintained in the new version: scalability in terms of the number of mobile nodes; scalability through data flow balancing; simplicity and flexibility; support for mobility and temporary disconnection (*handovers*);

support for high-performance communication; capacity for unicast, groupcast and broadcast communication and capacity for real-time monitoring of groups of mobile nodes.

ContextNet Core is made up of four independent micro-services. This architecture is reflected in the organization of this text, which has a section exclusively dedicated to each of these micro-services. The Core node that handles connection requests and communicates with the mobile nodes and these micro-services is called the Gateway, and has an exclusive section detailing it. There are other Core micro-services for load balancing the mobile connections among the Gateways, for handling temporarily disconnected mobile nodes and for sending context-aware group cast messages.

In the following text we present the related work; followed by sections on the design and implementation issues of each of the redesigned micro-services of ContextNet's Core; and finally we conclude and point to future work.

2 Related Work

The search for related work was mainly carried out by looking for middleware for mobile real-time communication that used the *Publish/Subscriber* communication protocol internally. The following briefly describes some of the works found, the theoretical references for these middleware systems, and the differences and similarities between them and the new ContextNet.

In [8], Kai Waehner lists the reasons why the telecommunications industry is migrating to solutions that use Apache Kafka. The author argues that telecommunications today is less about voice and increasingly about text (messaging, *e-mail*) and images (e.g. streaming video); and that the fastest growth is coming from (value-added) services provided over mobile networks.

Among the reasons listed in the article for using Kafka in telecom industry solutions are the need for real-time processing with scalability; and for a reliable system with zero downtime and no data loss. Traditional telecoms processes are separated into OSS (Operations Support System) - to which the mobile network is connected, BSS (Business Support System) and OSS-BSS-Integration; and according to the article Apache Kafka is used in all of them.

A generic framework for processing, storing and analyzing *IoT* data in real time, called *L-CoAP* and using Apache Kafka, was proposed in a 2015 article [9]. Unlike ContextNet, the *middleware L-CoAP* does not aim to allow real-time communication between its components, but rather to provide an abstract layer for accessing, consulting and controlling components *IoT*.

According to the authors [9], *L-CoAP* is the combination of *Cloud Computing* resources responsible for abstracting the limitations of *IoT* devices, with *IoT* middleware suitable for devices with resource restrictions.

The *L-CoAP* is made up of three different elements: a cloud; the *Smart Gateways*, designed to abstract and protect the *IoT* devices from external sources; and the end devices; one of the cloud components being an Apache Kafka instance. In this architecture, Apache Kafka is responsible for distributing the data flows received in the Cloud through the *Smart Gateways* in real time.

The architecture proposed by [9] does not address the problems of real-time mobile communication covered by ContextNet because it is aimed at communication with *IoT* devices. The work presents an architecture that could be incorporated into the Kafka layer of the new ContextNet or added as a new micro-service with several new functionalities for *IoT* components that do not exist today.

Article [10] describes *RADAR-base*, a *mHealth* (*Mobile Health*) medical data collection platform built on Apache Kafka. RADAR-base enables active and passive remote data collection; provides secure data transmission channels; and a scalable solution for storing, managing and accessing this data. The platform is currently used in the *RADAR-CNS* study to collect data from patients suffering from multiple sclerosis, depression and epilepsy. The real-time processing capacity of this platform is due to *Kafka Streams*. The mobile nodes connect via *Kafka-Proxy* to the *cluster* Kafka.

[10] also does not address the problems of mobile communication like ContextNet, but instead proposes a way of collecting and storing data. Integration via Kafka connectors is more restricted than ContextNet, and has been added to the new version as another connection alternative.

In a work on mobile clouds for smart cities [11] discusses selection criteria for cloud architectures to support Smart City applications. A mobile cloud solution is proposed based on the *cloudlet* approach. It uses *cluster Kafka* as a message-based *cloudlet* and *Kafka Proxy REST* to connect to mobile clients. The solution proposed by [11] for *Smart City* closely resembles the architecture of the new ContextNet, with the Gateways replaced by the *Kafka Proxy REST*.

3 Designing the ContextNet Kafka Core

This section will briefly introduce the main components/microservices of ContextNet Kafka Core (CKC), the new Version of the ContextNet Core, as well as their connections. The microservices will be detailed in the following sections.

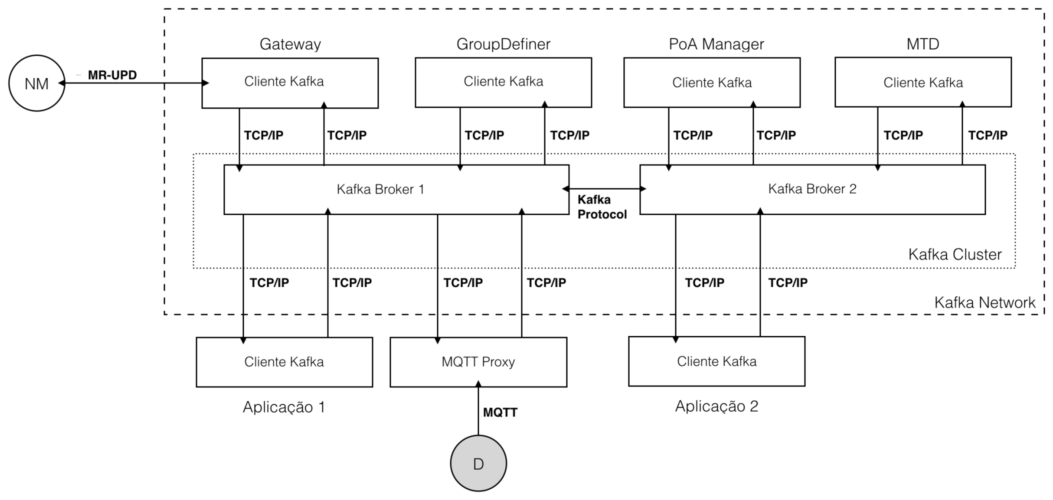


Figure 1: Overvew of the ContextNet Kafka Core

In Figure 1 the mobile nodes are represented by the "MN" element and connect to the CKC via the Gateway through an MR-UDP[12] connection. The Mobile Reliable

UDP (MR-UDP) protocol is the basis for the Gateway-mobile node interaction. This protocol implements TCP-like functionality at the top of UDP and has been customized to handle intermittent connectivity, Firewall/NAT traversal and robustness to changes of IP addresses and network interfaces. Each message, in either direction, requires an acknowledgment that, if not received, causes each transmission to be retried several times before the connection is considered broken.

The Gateway is a micro-service of the CKC, which is a Kafka Client, communicates via TCP/IP with one of the Kafka servers (*Brokers*). The *Kafka Brokers* make up the *Kafka Cluster*. The *Cluster* communicates via TCP/IP with the other CKC micro-services: *PoA Manager*, *Group Definer* and *Mobile Temporary Disconnect*, which are Kafka Clients.

The applications ("Application 1" and "Application 2") developed on the *middleware* are also Kafka Clients and also communicate with the *Cluster* via TCP/IP.

The *devices* that communicate exclusively via MQTT send data to the *Cluster* via a *MQTT Proxy* that communicates via TCI/IP with the *Brokers*.

3.1 Main Features

This section lists the main features and functionalities of the ContextNet Kafka Core. More details will be given in the following sections.

- *Unicast* in real time for mobile nodes and application nodes
- *Groupcast* in real time for mobile nodes (Dynamic groups defined from context information)
- *Broadcast* in real time for mobile nodes
- Collection of connection information
- Parameterized load balancing of Gateways
- Temporary storage of messages to disconnected mobile nodes
- Frequent (re)distribution to mobile nodes of a list of active gateway addresses sorted by availability
- Reception of data using the MQTT protocol
- Possibility of parallelizing Core applications

3.2 Scalability

One of the main objectives of the new ContextNet is to support scalability in terms of IoT/mobile applications and, consequently also in the number of connected mobile nodes.

Scalability in terms of the number of mobile nodes per gateway is also important, but it is limited by the mobile communication protocol used (in this case MR-UDP). Therefore, this work puts focus on making the static/stationary nodes of the ContextNet Core scalable to support a growing number of Gateways connected to the same Core, and therefore a growing number of mobile nodes.

The scalability of the static nodes not only indirectly allows for an increase in the number of mobile nodes, but also makes ContextNet more suitable for supporting complex applications that require a lot of processing.

Being scalable only in terms of the number of mobile nodes is limiting, since the nodes necessarily interact with applications that need to be able to support them.

The previous version of ContextNet did not have enough resources to allow the message flow of a given communication channel to be divided between several nodes, a feature that is now present in the new version.

With parallelization, it is hoped that the new ContextNet will be able to support applications that would fail without it; and that applications will be able to use this parallelization to improve communication performance.

4 Design of the Gateway for ContextNet Kafka Core

The Gateway is the main component of ContextNet Kafka Core, as It is through which all mobile nodes communicate with static nodes. On the one side, it accepts mobile connections of, and communicates using the MR-UDP protocol, and on the other side, uses Kafka protocol and translates from one protocol to the other. Gateways support unicast messages from the mobile nodes to the static nodes and vice versa; support groupcast messaging - synchronized or not - from the mobile nodes and the static nodes to the mobile nodes; send broadcast messages from the static nodes to the mobile nodes; collect and send data on the state of the hardware and mobile connections; and identify messages not delivered to the recipient mobile nodes and forward them to a repository (MTD).

The Gateway of the new version of ContextNet Core, in addition to the features listed above, is able to publish *Records* for various topics according to the node's address; collect the response time for *Ping* messages (used for latency measurement); reconfigure the size of the waiting windows for responses to *Ping* messages and maintain a *mini-buffer* with the last messages sent by a mobile connection. In the remainder of this section project decisions regarding the Gateway and theirrationale will be presented.

4.1 Checking latency and disconnections of mobile connections

Data on mobile connections is collected by sending *Ping* messages. The Gateway forwards these messages to the mobile nodes connected to it and the mobile nodes respond to this message with a *Pong* message. This is how communication latency is measured.

This latency could be processed by either the Gateway or the PoA Manager. Since sending messages necessarily goes through the Gateway, it's natural to delegate this task to it. However, the Gateway is already the most overloaded element in terms of processing, so delegating this task to the PoA Manager significantly reduces the load in the Gateways. And in fact, the increase in the amount of data traveling through the Core under this architecture, which could be seen as a disadvantage, is not significant. So, the PoA Manager was chosen to handle the processing the latency.

In the former ContextNet, this accounting was carried out by the Gateway, and consisted only of counting the number of connections whose latency was above a pre-established limit. This information was only consumed by application nodes connected to the Core, and had no influence on load balancing. The new architecture allows latency and other connection metrics to be taken into account by the balancing algorithm.

The process of collecting *Pongs* happens within time windows. These windows start after the *Pings* have been sent, and while the time window is open, the moment at which *Pong* messages are received is recorded. *Pong* messages received after the end of the window are discarded. These records of *Pong* arrival times are sent to the PoA Manager - so that it can process the latency.

We decided to use windows rather than continuous collection of *Pongs* because the second option would require the Gateway to maintain an updated structure in real time of the *Pings* sent and the *Pongs* received by each mobile node, increasing its complexity and processing cost with a gain in accuracy that was not considered significant.

In the new version of the ContextNet Core, the frequency of sending *Pings* and the length of the windows for receiving *Pongs* are defined by the PoA Manager and sent to the Gateway via a "*PingConfig*" configuration topic. On receiving a "*PingConfig*" message, the Gateway reconfigures its process for sending *Pings* and collecting *Pongs*. Introducing this flexibility on the window length allows it to better adapt to current network conditions and allows these conditions to be better evaluated, increasing accuracy. Assuming that for a given window the percentage of mobile nodes that replied to the Ping on all gateways is very low, then by increasing the window size enables us to check whether the low percentage is due to disconnections or to high latency connections. The same could also be verified with the continuous recodring of the *Pong*, arrivals but at a much higher processing cost.

4.2 Handling Message Sending Failures

When there is a problem in the connection between the Gateway and a mobile node that prevents messages from being properly delivered; and the Gateway, unaware of the problem, tries to send a message through this same connection, there may or may not be a *failure to send*, generated by the MR-UDP protocol. In the case of a failure to send, an exception is generated, caught and the message encapsulated in a *Record* and sent to *Mobile Temporary Disconnect - MTD* by the Gateway via the "*UnsentMessage*" topic. If there is no *failure to send* and the message has not been sent, the MR-UDP protocol immediately notifies the Gateway of the failure. This notification, however, does not tell the Gateway which message was not sent.

To enable the unsent message to be informed to the MTD, the new version of the Gateway maintains a *mini-buffer* of messages per connection. This *mini-buffer* keeps in memory the last two messages sent without failure by the Gateway for each connection (each connection connects a single mobile node to the Gateway). If the connection notifies the Gateway that it has failed to send, the messages in the *mini-buffer* are sent to the *MTD* microservice. This *mini-buffer* guarantees that all messages will be sent at least once to their recipient nodes.

An environment variable can be used to determine whether or not to use the *mini-buffer*. Its use guarantees that all messages will be delivered at least once to the recipients, but also increases the execution time of the *thread* responsible for sending messages. This *thread* updates the *mini-buffer* at the end of each successful send, and this update causes it to take longer to finish, and hence causes it to remain longer in the Gateway's *Thread Pool*. In addition, updates to the *mini-buffer* that concern the same MR-UDP connection cannot occur in parallel, which causes the mini-buffer update time to increase significantly if several messages are sent in a burst over the same connection. The implementation of *mini-buffer* can therefore slow down the Gateway and it is left to the developer to decide whether or not to adopt it.

4.3 Possibility of sorting messages

Kafka would allow message sorting to be extended to all types of communication, however, this would require a lot of processing from the Gateway, and would significantly decrease its performance, so it was not added to the new version. Only communication between applications allows message sorting.

4.4 Project Pattern: Thread Pool

The Gateway was developed using the *Thread Pool* design pattern. During the tests, it was observed that the ideal size of the *Thread Pool* cannot be estimated without first knowing the number of mobile connections that the Gateway will need to support. We therefore decided to allow the developer to define the size of the size of *Thread Pool* according to their needs. There is a default size, but if necessary the developer can redefine the size of the *Thread Pool* via an environment variable.

5 Design of the PoA Manager for ContextNet Kafka Core

The PoA Manager is the microservice of the Core responsible for load balancing mobile connections among all instances of the Gateway; and for producing an up-to-date list of the addresses of the active/available Gateways (and theirs load), for new mobile nodes to connect. This load balancing is performed by the mobile nodes individually reconnecting themselves to "lighter loaded" Gateway. This occurs after reception of a *handoff* messages from the PoA Manager informing the current ranking of light to heavy loaded Gateway instances.

As the PoA Manager only consumes periodic reports from the Gateways, the data flow it receives from Gateways is is rather small compared to that of the other Core components. Hence, it can dedicate itself to heavier statistical processing. In this respect, the new version of PoA Manager extends the functionality of the former PoA Manager and adds new ones.

In the new version, the Gateway instance ranking algorithm can be defined by the ContextNet Core operator/deployer; the parameters of the Gateway's *Pings* sending routine are determined by the PoA Manager and the *Pongs* receiving windows can be dynamic and are also parameterized by the human deployer; latency accounting produces statistical data on connections; the active Gateway checking routine has been redesigned and can also be parameterized by the deployer; and the load balancing algorithm has been entirely redesigned.

The following sections describe project decisions and the flow of management/ performance *records* among the PoA Manager and the active Gateways. The following figure gives an overview of the PoA Manager processes and how they are linked. In the following sections, each of them will be described in more detail.

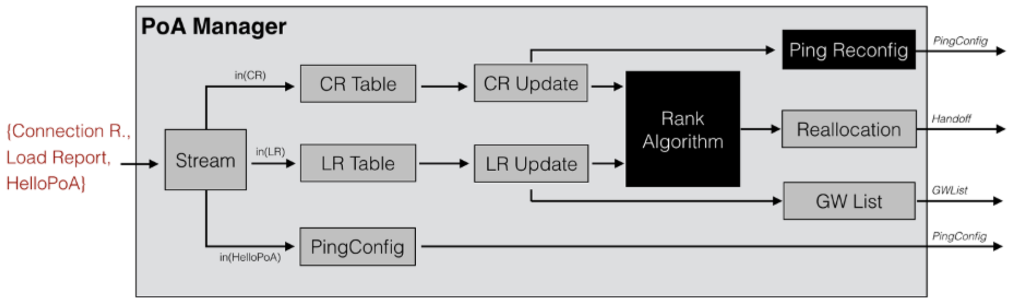


Figure 2: Visão Geral do PoA Manager

5.1 Initial configuration of the *Ping-Pong* routine

When a new Gateway joins the ContextNet Kafka Core, it informs the PoA Manager of its arrival via a *record* "*HelloPoA*". On receiving a "*HelloPoA*", the PoA Manager informs the new Gateway of the Initial frequency of sending *Pings* messages, and the windows for receiving *Pongs* via a *record* "*PingConfig*" addressed to the new Gateway. This configuration is represented in the 2 by the flow that starts in "*Stream*" with the receipt of records, and moves to the "*PingConfig*" process.

5.2 Gateway Load Ranking Algorithm and Load Balancing

Gateway load balancing, represented in the 2 by the process "*Reallocation*", happens after the *Gateway Load Ranking Algorithm (GLR)* of the PoA Manager has analyzed all the Gateways'r reports and has assigned scores to each that reflect their current connectivity load. This GLR algorithm can be implemented by the developer/operator of the ContextNet Core. Or else, they can choose to use one of the two GLR algorithms already available, which will be described later in this text.

The (GLR) ranking algorithm receives as inputs the data about mobile/ MR-UDP connections - in 2 from the "*CR Update*" process; and the data on loads on active Gateways - in 2 from the "*LR Update*" process. It returns as output a *map* data structure containing the active Gateway IDs as keys, and their respective load scores as values. The higher the score of a Gateway, the more lighter is its load, and hence the more suitable it is to receive new connections.

As depicted in Figure 2, the load balancing process receives the map of all Gateways' scores as well as well as their connection data. Based on this information, it calculates how many nodes should ideally be connected to each gateway. This calculation is represented in the following formula where g is the total number of Gateways; $W(X)$ is the score of Gateway X; n is the total number of nodes; and $In(X)$ is the ideal number of mobile nodes connected to Gateway X.

$$In(X) = nW(X)/(\textstyle\sum_{i=1}^{g} W(i))$$

After calculating the optimum number of nodes connected to each Gateway, reallocation is initiated and reconnection *"handover"* messages are generated and sent to various mobile nodes connected at overloaded Gateways; these reconnections are aimed at bringing the Gateways to at the previously calculated number of ideal

An efficient way to avoid requesting *handoffs* to a same mobile nodes in very short intervals of time is still being developed. This is important to avaoid high "handover churn due to frequent variations of the Gateways scores obtained by the GLA within PoA Manager.

5.3 Processing of *Load Reports* from the Gateways

Gateways periodically send data about the (hardware) resource usage to the PoA Manager via the "*LoadReport*" topic. The *LoadReport* uploads the following data: utilization rate and size of the *Thread Pool*; CPU load rate; total memory and fraction available; total Java virtual machine memory and fraction available.

The periodicity of sending this *LoadReport* is fixed and uniform for all Gateways and well known by the PoA Manager. In the 2 the "*LoadReport*" arrive at the PoA Manager via the "*Stream*" process, and are stored in a *Table* - represented by the "*LR Table*" process. The arrival times of these topics are also recorded.

Periodically the PoA Manager starts the "*LR Update*" process that is responsible for de-serializing the content of the topics and checking through the arrival logs which Gateways are active. A Gateway is considered inactive if no "*Load Report*" has been received from it in the last X windows; where X is informed by the developer and the window size is equal to the period in which the Gateway sends "*Load Reports*".

After checking which Gateways are active, the PoA Manager notifies the other elements (microservices) of the core via the "*GWList*" - "*GW List*" process of the 2 - which Gateways these are and what their address is. This topic is received by the Gateways and forwarded to all the mobile nodes. At the end, the "*LR Update*" process also feeds the ranking algorithm with load data on the active Gateways, as shown in Figure 2.

5.4 Processing of the *Connection Report* from Gateways

The "*ConnectionReport*" holds data on each Gateway's mobile connections and reaches the PoA Manager via the "*Stream*" process represented in Figure 2 and is stored in a *Table* - represented by the "*CR Table*" process. These Kafka topics are processed in the "*CR Update*", a periodic process responsible for de-serializing the topics in the "*CR Table*" and accounting for latency. The "*ConnectionReport*" contains data about the Gateway's mobile connections, including: the time the *Ping* message was sent to the mobile nodes; the list of the mobile nodes' response times; and a list of all the mobile nodes.

Based on the data sent via the "*ConnectionReport*" (the time the *Ping* message was sent to the mobile nodes; the list of times when the replies from the mobile nodes arrived; and the list of all the mobile nodes), the metrics called "*Connection Statistics*" is calculated, referring to the mean communication latency of each Gateway. Latency means the round-trip time between sending the *Ping* and receiving the *Pong*.

The "*Connection Statistics*" enconcompasses the mean, standard deviation, variance, median, maximum and minimum latency values; and the percentage of nodes whose *Pong* was received within the window - "*Ping Rate*". This information about the connections and inputs are generated in the "*CR Update*" feeds - as can be seen in Figure 2 - the ranking algorithm.

This information is also used in the "*Ping Reconfig*" procedure, which is responsible for assessing whether there is a need to change the duration of Ping windows, and if so, what the new size should be. This process can be disabled and parameterized by the developer. Based on the "*Ping Rate*" of each Gateway, the process calculates the total "*Ping Rate*" of the metric network used to calculate the change. For each "*Ping Rate*" interval, the developer tells you by which factor you should multiply the duration of the window. If this factor is one, the window remains the same.

6 Design of the Mobile Temporary Disconnect Service for ContextNet Kafka Core

The new MTD service has all the features of the previous version plus the automatic discarding of messages. It stores messages that could not be sent to the recipient mobile nodes due to failed connection between them and a Gateway. As explained in the section 4, these unsent messages are reinserted into the Core in the "*UnsentMessage*" Kafka topic. MTD collects all these *Records* and stores them until it notices, by reading a *"ConnectionReport"* that the recipient has reconnected. When it notices the reconnection, MTD dispatches the *Record* to the proper Kafka topic of Core, and addressed to the recipient.

6.1 Policy for discarding messages after some timeout by MTD

In the MTD of the former Core, unsent messages were stored indefinitely in the MTD. In the new MTD, messages are discarded after "X" seconds, where "X" is entered by the sender of the message or a *default* value defined by the developer. The developer can also choose to standardize "X" for all messages.

This policy of discarding messages not only reduces the amount of main memory the MTD needs to operate, but also prevents messages (whose delivery no longer makes sense due to the long timespan of the disconnection) from being sent and unnecessarily occupying memory space in the ContextNet Core and mobile node memory. Implementing this policy makes the MTD more complex and requires more processing to operate. It was felt that the functionality gained would compensate the increase in processing.

7 Design of the Group Definer for ContextNet Kafka Core

The main function of the Group Definer (GD) is to classify mobile nodes into different groups based on their context information sent by them, and support groupcast messaging to mobile nodes that happen to be in/ have the same context [13].

To use this microservice, the ContextNet Kafka Core application developer just needs to implement the *GroupSelection* interface, the component responsible for the grouping logic.

In the new ContextNet, the amount of data flowing through the Group Definer has been drastically reduced compared to the previous version, since in the new version the GD only consumes *"ContextReport"* topics. The new version now allows core nodes executing GD with the same logic to be parallelized if necessary; and for GD nodes with different logics to coexist in the same Core.

In the new ContextNet, all messages received by the Gateway from mobile nodes will contain context information. This information is placed by the Gateways in the *Record "ContextReport"* and consumed by the Group Definer. Using this information, the selection algorithm implemented in the Group Definer decides which groups the mobile node belongs to, and relays this information to all Gateways via a *Record "GroupAdvertisement"*. Gateways consume the *"GroupAdvertisement"* and use this information to generate the groupcast messages addressed to all its conneced mobile nodes that pertain to the group.

As it was the case with the previous CD version, the application developer must implement an Interface to make use of the Group Definer. This interface is responsible for receiving context information from the mobile nodes (through the context topic) and using it to apply some logic to define which group or groups a the mobile node belongs to.

7.1 Project Decisions

The new ContextNet Core allows two implementations with different Group Definer logics to coexist. In the previous version, each *"GroupReport"* for a given mobile node that reached the Gateway disregarded the information from the former *"GroupReports"*. In the new version, the Group Definers have identifiers that are transmitted as keys to the *"GroupReports Records"*. Through these identifiers, the Gateway is informed of the origin of each *"GroupReport"* received, and keep in memory the information from the last *"GroupReport"* sent by each Group Definer. This possibility of two different instances of Group Definer coexisting means that two applications with different node grouping logics can coexist.

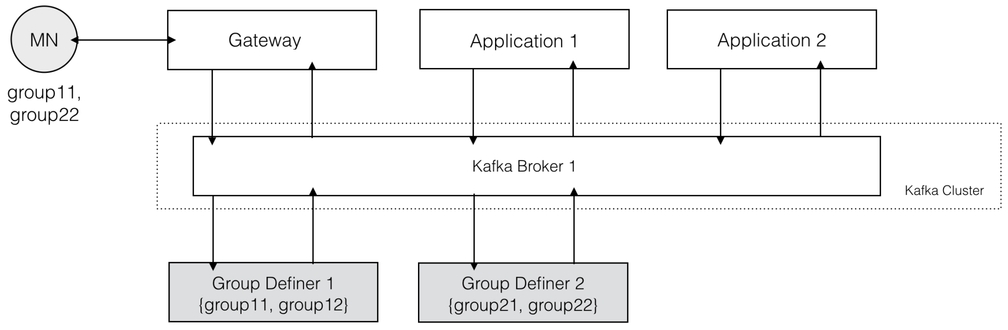


Figure 3: Filas GD

Figure 3 illustrates this situation. It shows two Group Definer implementations (*"Group Definer 1"* and *"Group Definer 2"*) capable of classifying mobile nodes into four groups: *group11*, *group12*, *group21* and *group22*.

The *"Group Definer 1"* implementation complies with *"Application 1"* and classifies the nodes in the *group11* and *group12* groups; and the *"Group Definer 2"* implementation complies with *"Application 2"* and classifies the nodes in the *group11*, *group12* groups. In this configuration, there may be nodes such as "MN" from Figure 3 which belong to *group11* and *group22* and which receive *groupcast* messages addressed to both groups.

8 Performance Tests

The tests designed for ContextNet Kafka Core are divided into two groups: those aimed at evaluating performance and those aimed at evaluating the effectiveness of certain components. Performance is understood to mean the ability of ContextNet components to support effective real-time communication in scenarios with a high volume of messages exchanged and a large number of connected mobile nodes. In other words, performance tests assess how scalable communication is in ContextNet Core.

Among the performance tests are those aimed at evaluating the impact of parallelization on the performance of static nodes and those aimed at evaluating the performance of the Gateway.

The following section describes the test infrastructure and the remainder describes the tests designed, their objectives, the setup and the metrics collected.

8.1 Computing Infrastructure used

All the tests were carried out on virtual machines of the *Cloud-DI*. A total of eight virtual machines with two processors were used, five with 16 GB of RAM and three with 24 GB of RAM. All the virtual machines are logicaly connected in a bus topology.

The Gateways were allocated to the machines with 24GB of RAM because they are the components with the lergest need for RAM due to the high number of MR-UDP connections.

Empirically, it was noted that these connections take up a lot of memory space, especially at the moment of MR-UDP connection set-up. Therefore, the more RAM available to the Gateway, the more mobile connections it can establish.

In almost all the tests, the following distribution is used: three 16GB virtual machines are used to simulate the mobile nodes; one 16GB machine houses the Kafka structure; and one 16GB machine houses the application processing and the other ContextNet Kafka Core micro-services (Group Definer, MTD and PoA Manager). In the tests in which there is a different distribution of virtual machines, this will be described specifically in the subsection.

All the ContextNet Kafka Core components have been dockerized to make it easier for any developer to implement them in the test environment and repeat the experiments. All the images used in the tests are published in the LAC repository and will be identified in the following subsections.

For all the tests described in the sections below, 5 experiments were carried out.

8.2 Configuration of the Simulated Mobile Nodes

To carry out the tests described below, various quantities of mobile nodes (MNs) were simulated. The way in which the MNs simulation is carried out directly affects the obtained results . Therefore, before initiating the tests, explorarory experiments were carried out with several different configurations of mobile nodes in order to assess which would be the most suitable for the following performance tests.

With "configurations"we mean the layout of the containers that simulate the mobile nodes, as well as the simulated mobile node itself and the time interval between subsequent connection requests from all mobile nodes.

The mobile nodes were divided into groups of equal size. Each group is simulated using a Java executable. An image of this executable was created and published. This image is executed in a container during the simulation.

The experiments used the *round-trip time* of the messages sent from the MNs (to an application node in the Core) as a metric. We were looking for a configuration that would best emulate the behavior of the MR-UDP protocol, and that would have minimal influence on the *round-trip time* measurements.

It was observed that the larger the number of containers, the shorter the *round-trip time*, and the number of containers is limited by the machine's RAM memory. The size of the containers does not increase proportionally with the increase in the number of simulated mobile nodes running inside them, as each container runs a Java virtual machine whose size is not solely related to the number of simulated mobile nodes it runs. The largest number of containers supported by a virtual machine was 40, so 40 containers were used in all the simulations.

In the configuration we finally adopted for simulation, each mobile node has n threads in a *Thread Pool* at its disposal. As the total number of simultaneous threads on a machine is limited, n is also limited. It has been observed that the higher the n, the lower the *round-trip time.* The largest n supported by a machine was 4, so each simulated MN had 4 threads at its disposal.

It was also observed that the Gateway has a limit on the number of connections it can support simultaneously per time. MR-UDP connections consume a significant amount of memory to be established, so when too many are established simultaneously, the Java virtual machine runs out of memory and starts dropping existing connections. In order to make the process of connecting the mobile nodes to the Gateway possible, a minimum interval was set between the establishment of two subsequent connections.

Experiments were carried out to find the shortest possible interval between connections, and a value of 1000 milliseconds was identified. As in the simulations we have 40 independent containers connecting simultaneously to the Gateway, for the average interval between two connections to remain at 1000 milliseconds, the interval between two connections requested by simulated mobile nodes from the same container would have to be 40000 milliseconds.

8.3 Scalability tests for the basic Core Inbound. communication

The basic/simplest communication tests are performance tests based on one of several scenarios. In the basic tests, the MNs start communicating with a connected Gateway by sending messages to an application node in the Core, and remain connected to th Gateway throughout the experiment by sending messages to the application node at a constant rate. This application responds to with an ack to all messages received from the mobile nodes. An MTD node is also deployed in the Core.

The metrics used in these tests are the *round-trip time* (RTT) of the messages sent by the mobile nodes and their delivery rate (TE). Both RTT and TE are measured by the process that controls the mobile nodes.

8.3.1 Scenario 1

Scenario 1 is the base scenario for all the other tests. Its results will be the benchmark for all other of the scenarios that will be evaluated. The tests in this scenario aim to verify how fast (RTT) and reliable (TE) the mobile node - Core - mobile node communication is at a given message sending rate (TM) for different numbers of nodes (QTD); number of Kafka brokers (BK); and number of Gateways (GW).

In the tests for this scenario, we varied the number of mobile nodes (MNs) from 1000 to 5000; the number of Kafka brokers (1, 3) and consequently the number of application nodes replying to messages from the nodes; and the number of Gateways (1, 3).

The message sending rate by the MNs chosen was 4 per minute per node, rates also used in [1] to evaluate the performance of the original ContextNet Core. In addition to being a realistic rate for IoT applications, its adoption allows for a fairer comparison between the results of the new ContextNet and the original one.

The observation period began when all the MNs were connected to the Gateway and ended when every mobile node had sent 500 messages to a node iexecuting within the ContextNet Kafka Core.

At the end of this test, we will assess how much each of the simulation variables influences RTT and TE; we will also assess whether the performance of the new ContextNet for Inbound communication is satisfactory; whether Inbound communication is reliable; whether performance remains constant over time; and we will compare the previous and this new version of the ContextNet Core. We consider RTTs of less than 1000 milliseconds to be satisfactory.

8.4 Scalability and load balancing tests for Core Outbound Communication

The basic communication tests are performance and functionality tests, and were divided into three scenarios. The first (Scenario 2) deploying only "stationary" simulated MNs and without using the MTD micro-service; the second (Scenario 3) with MNs that disconnect from and reconnect at Gateways with a certain frequency, and still without use of the MTD; the third (Scenario 4) with mobile nodes that disconnected and reconnected at a certain frequency and with support of the MTD micro-service.

In all scenarios, the MNs start communicating with the Gateway by sending a message to a dummy application node connected to the Core. After receiving this initial message, the application node send messages to the MN at a certain rate (TM). An MTD node can also be connected to the Core.

The metrics used in these tests are the round-trip time (RTT) of the messages sent by the application to the MNs and their delivery rate (TE). Both RTT and TE are measured by the processes that controls the application.

At the end of this test we evaluated whether the performance of the new ContextNet for Outbound communication is satisfactory; we compared the performance of Inbound and Outbound communications; we evaluated how much MTD makes communication more reliable and how much it influences performance.

For all scenarios, the message sending rate chosen was 4 per minute per node, the same as in Scenario 1, to facilitate comparisons.

The observation period began when all the nodes were connected to the Gateway and ended when the application had sent 500 messages.

8.4.1 Scenario 2

Scenario 2 is the baseline for all the other tests, based on the results of which some of the scenarios listed in this section will be evaluated.

Measuring RTT and TE, the tests in this scenario aim to verify how fast and reliable the *Core-MNs-Core* round-trip communication is at a given message sending rate (TM) for a given number of MNs (QTD); certain numbers of Kafka brokers (BK) and of Gateways (GW).

In the tests for this scenario we used 5000 mobile nodes (MNs); a Kafka broker and consequently an application node in the Core sending messages to the nodes; and a single Gateway.

Scenario 2 will also be compared to Scenario 1. Both have the same metrics and the same variables, but deal with different types of communication. This comparison aims to identify if there is a difference in performance between Inbound and Outbound communication in the ContextNet Kafka Core.

8.4.2 Scenario 3

The results of Scenario 3 will be compared with the results of Scenario 2. Both have the same metrics and the same variables.

The tests in this scenario aim to verify how fast (RTT) and reliable (TE) the *Core-MNs-Core* round-trip communication is in an environment where there are errors/faults in the network connecting the MNs to the Gateway. To simulate these communication errors, a portion of the mobile nodes (i.e., DES) disconnects and reconnects from/to a Gateway at a certain frequency.

In the tests for this scenario we used 5000 MNs; a Kafka broker and consequently an application node sending messages to the nodes; and tested two percentages for DES (30% and 70%). At each disconnection, each MN waits 15,000 milliseconds until it reconnects to the same Gateway.

In this test, we evaluated the impact of these reconnections on RTT and TE by comparing their results with those of Scenario 2.

8.4.3 Scenario 4

The results of Scenario 4 will be compared with the results of Scenarios 3 and 4. Both have the same metrics and the same variables. The MTD is enabled in this scenario.

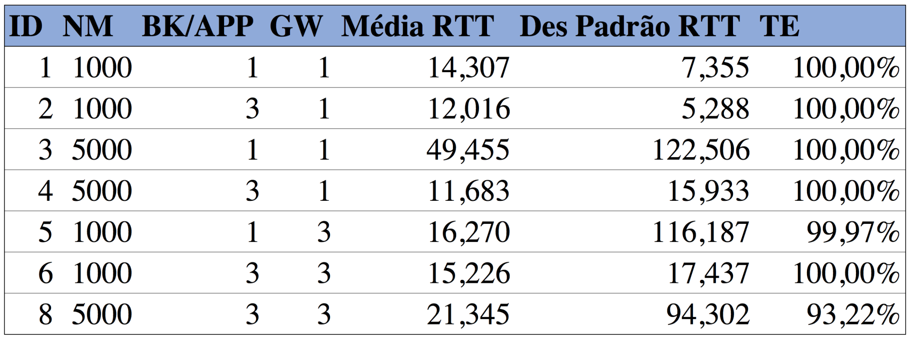
Comparing the results of Scenario 4 with Scenario 4 we can check MTD effectiveness and the impact in RTT and TE.

8.5 Results

The following subsections present the results of the tests described above. All time units shown are in milliseconds.

8.5.1 Results of Scenario 1

For Scenario 1, eight different configurations were selected, varying the number of nodes (NMs), gateways (GWs), applications and Kafka brokers (BK/APP). One of these configurations (i.e., 5000 MNs, 3 GWs, 1 BK/APP) couldn't be executed because several MR-UDP connection failures occurred during the execution attempts.



ID	NM	BK/APP	GW	Média RTT	Des Padrão RTT	TE
1	1000	1	1	14,307	7,355	100,00%
2	1000	3	1	12,016	5,288	100,00%
3	5000	1	1	49,455	122,506	100,00%
4	5000	3	1	11,683	15,933	100,00%
5	1000	1	3	16,270	116,187	99,97%
6	1000	3	3	15,226	17,437	100,00%
8	5000	3	3	21,345	94,302	93,22%

Figure 4: Results of Scenario 1

Figure 4 shows a summary of the test results for Scenario 1. Each configuration is identified by an ID (first column of Figure 4) which will be used throughout the section to identify them. For each of the configurations, Figure 4 shows the average (Mean RTT) and standard deviation (Standard Deviation RTT) of the round-trip time of all the messages carried; and their delivery rate (TE).

Analyzing Figure 4, it can be concluded that the performance of the new ContextNet for Inbound communication is satisfactory, since in its worst performance the average RTT is 49.45 milliseconds, considerably less than 1000 milliseconds; and that the average RTT for all configurations is 20.04 milliseconds.

It can also be concluded from Figure 4 that ContextNet is extremely reliable for scenarios with up to 5000 MNs (mobile nodes) per Gateway. The TE was 100% in 5 of the 7 configurations tested and was above 99.9% in another. In configuration 8, the only one with 15,000 MNs connected to the system, the TE decreased to 93.22%, which may indicate that in configurations with a larger number of MNs, confidence in ContextNet Kafka Core decreases. It can be seen that only the number of MNs has a direct influence on TE.

As shown in Figure 4, comparing the RTT of configurations 3 and 4, it can be concluded that the parallelization of the application (which was made effective in 4) can contribute to an improvement in performance. The average RTT in 3 is 49.45 milliseconds and the standard deviation is 122.50, while in 4, with the same number of mobile nodes (5000), the average RTT is 11.68 milliseconds and the standard deviation is 15.93.

Since both the mean and standard deviation of the RTT of configurations 1 and 2 are very close, in order to compare the performance of configurations 1 and 2 we also looked at the histograms in Figures 5 and 6 and analyzed Figure 7.

The histograms in Figures 5 and 6 show the distribution of the average RTT for each node. From 5 we know that the average RTT of the vast majority of nodes in configuration 1 is between 12.5 and 15 milliseconds; while from 6 we know that the vast majority of nodes in configuration 2 is between 10 and 12.5 milliseconds. Hence we can conclude that communication is faster for nodes in configuration 2 than for nodes in configuration 1.

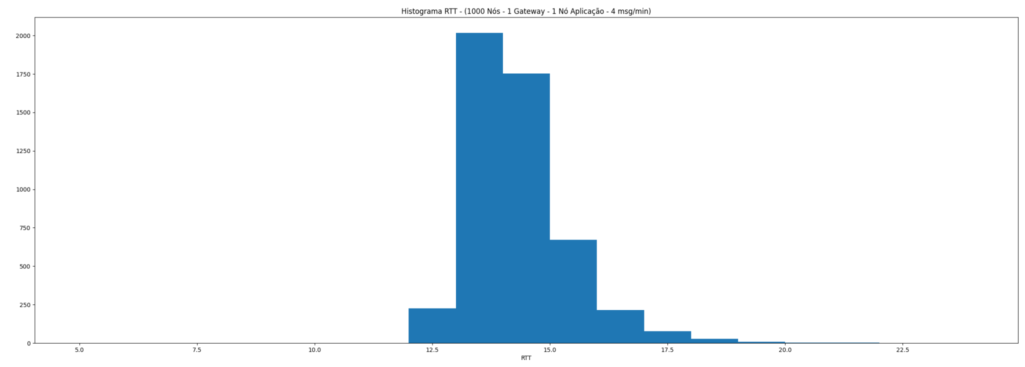


Figure 5: Histogram of Configuration 1

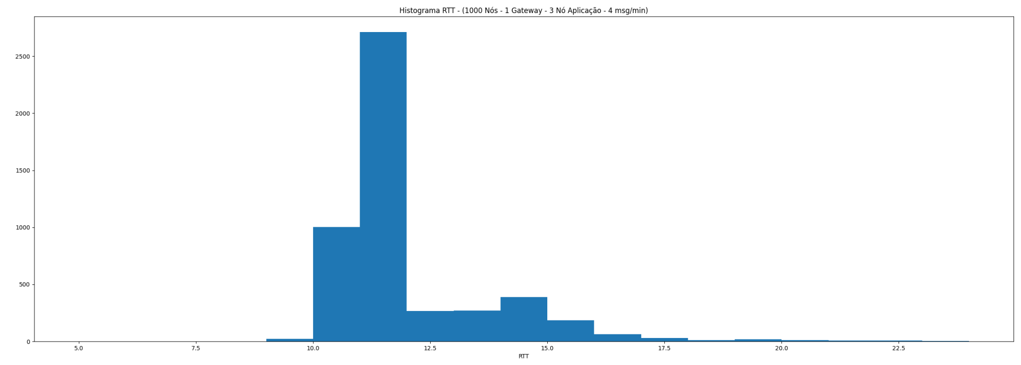
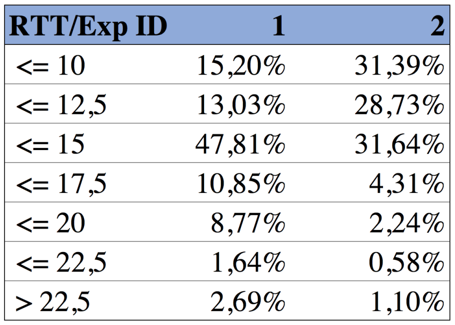


Figure 6: Histogram of Configuration 2



RTT/Exp ID	1	2
<= 10	15,20%	31,39%
<= 12,5	13,03%	28,73%
<= 15	47,81%	31,64%
<= 17,5	10,85%	4,31%
<= 20	8,77%	2,24%
<= 22,5	1,64%	0,58%
> 22,5	2,69%	1,10%

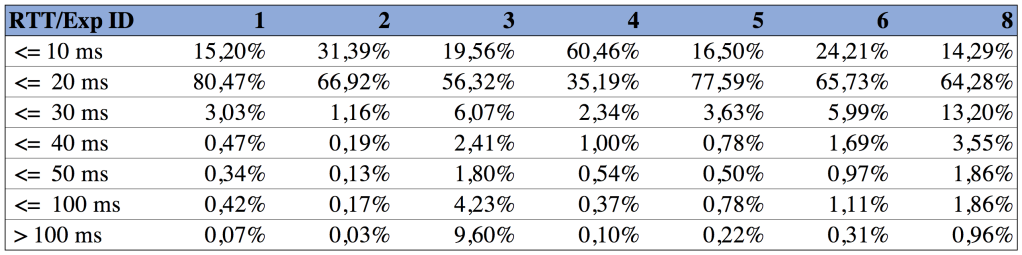
Figure 7: Compating Configurations 1 and 2

Figure 7 shows the distribution of all RTTs for configurations 1 and 2, shown in units of 2.5 milliseconds each. Figure 7 shows that 15.2% of the RTTs in configuration 1 were shorter than 10 milliseconds, and 31.39% of the RTTs in configuration 2 were shorter than 10 milliseconds. While in configuration 2 60.12% of the RTTs were shorter than 12.5 milliseconds, only 28.23% were shorter than 12.5 milliseconds in configuration 1. Only 8.24% of the RTTs in configuration 2 were longer than 17.5 milliseconds, while in configuration 2 23.96% of the RTTs were longer than 17.5 milliseconds.

From these observations we can conclude that the performance of configuration 2 is superior to that of configuration 1, and given the characteristics of the configurations analyzed, this superiority can be attributed to the parallelization of the applications.

From the impossibility of running configuration 7 with 5000 MN, 3 GW, 1 BK/APP shown above; and from the success of running configuration 8, it can be concluded that application parallelization (the only difference between configurations 7 and 8) can be a determining factor in improving ContextNet Kafka Core's performance; and even in the viability of executing some scenarios.

The above conclusions about the comparisons between configurations 1 and 2; and 7 and 8 corroborate the conclusions about the comparison between configurations 3 and 4.



RTT/Exp ID	1	2	3	4	5	6	8
<= 10 ms	15,20%	31,39%	19,56%	60,46%	16,50%	24,21%	14,29%
<= 20 ms	80,47%	66,92%	56,32%	35,19%	77,59%	65,73%	64,28%
<= 30 ms	3,03%	1,16%	6,07%	2,34%	3,63%	5,99%	13,20%
<= 40 ms	0,47%	0,19%	2,41%	1,00%	0,78%	1,69%	3,55%
<= 50 ms	0,34%	0,13%	1,80%	0,54%	0,50%	0,97%	1,86%
<= 100 ms	0,42%	0,17%	4,23%	0,37%	0,78%	1,11%	1,86%
> 100 ms	0,07%	0,03%	9,60%	0,10%	0,22%	0,31%	0,96%

Figure 8: Comparison of the Configurations for Inbound Communication

Based on the observations in Figures 4 and 8, we can compare configurations 5 and 6. Their average RTTs are very close, but the standard deviation of configuration 5 is significantly higher, meaning there is a greater dispersion of the RTT. Figure 8 shows the distribution of all the RTT for all the configurations in 10 and 50 millisecond units/bands. Through Figure 8 we can see that there are no significant differences in the distributions of configurations 5 and 6.

To check whether increasing the number of Gateways influences performance, we compared pairs of configurations, i.e. configuration 1 with configuration 5; and configuration 2 with configuration 6. By 4 we know that 1 and 5; and 2 and 6 have very close RTT average values, with the configurations with more than 1 Gateway having slightly higher RTT averages. In 8 we see that the RTT distributions in bands are very similar between the configuration pairs 1 and 5; and pairs 2 and 6. From these observations it can be said that increasing the number of Gateways does not in itself significantly impact on ContextNet's performance.

To check whether ContextNet's performance remains constant over time, Figures 9, 10, 11, 12, 13, 14 show the average RTT per message sending activity. The sequence order varies from 0 to 500, as 500 messages were sent per node.

Observing Figures 9, 10, 11, 12, 13, 14, it can be seen that ContextNet Kafta Core's RTT performance does not Deteriorate over time but remains almost constant.

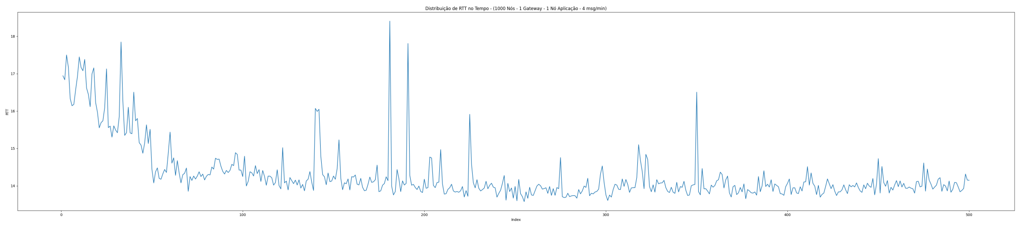


Figure 9: RTT Average Distributed over Time - Configuration 1

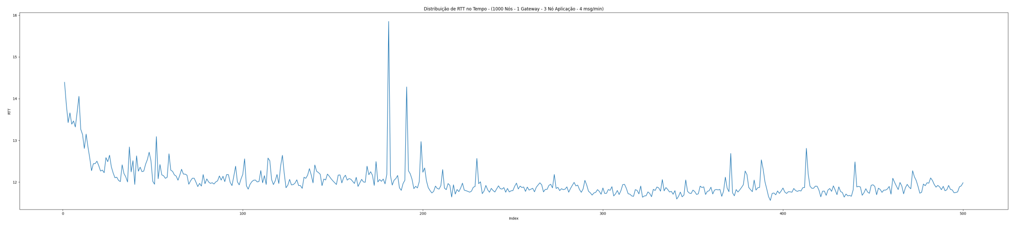


Figure 10: RTT Average Distributed over Time - Configuration 2

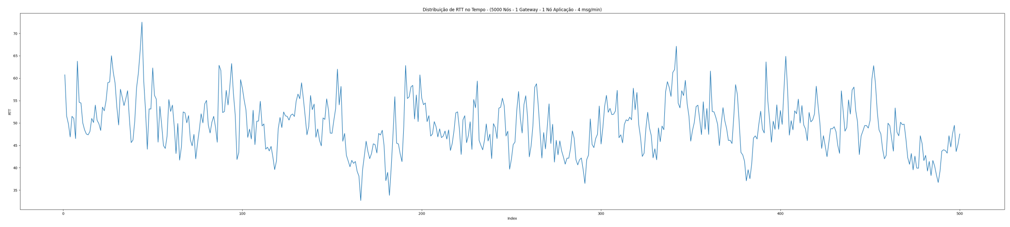


Figure 11: RTT Average Distributed over Time - Configuration 3

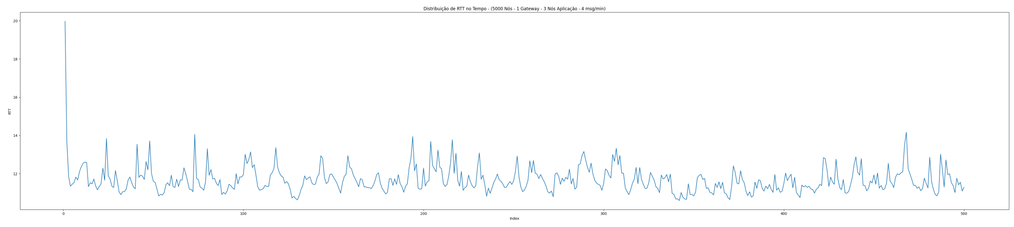


Figure 12: RTT Average Distributed over Time - Configuration 4

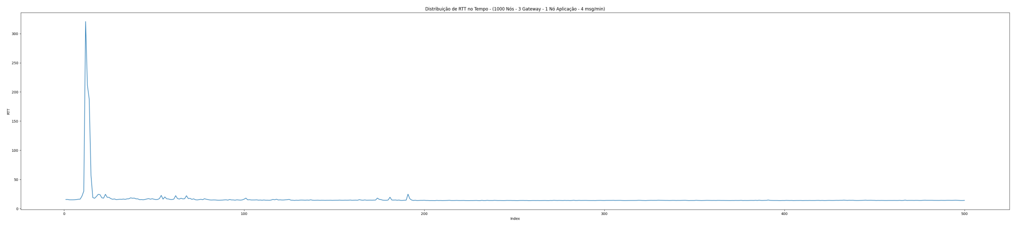


Figure 13: RTT Average Distributed over Time - Configuration 5

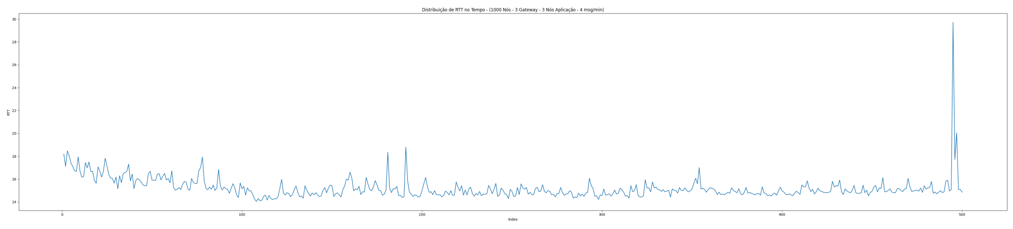


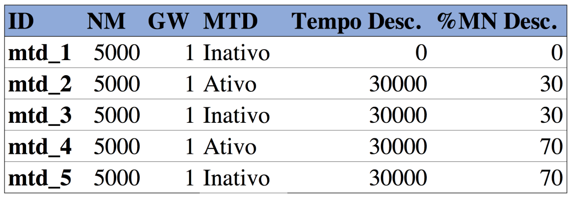
Figure 14: RTT Average Distributed over Time - Configuration 6

In [5] the performance of the previous ContextNet is documented. The RTT varied between 20 and 45 milliseconds, and was considered satisfactory. Therefore, it can be concluded from Figure 8 that the performance of the new ContextNet compared to the old one is satisfactory. Unfortunately, the related test scenarios used for evaluating the former and the test scenarios of this new ContextNet versions were not directly compared, as there is no way to make a direct comparison between totally different simulations.

Given the above, we conclude that the Inbound communication of the new ContextNet is reliable; that its performance is satisfactory, remains constant over time, and is comparable to that of the old ContextNet. We also concluded that the parallelization of applications can improve performance in certain situations; that increasing the number of mobile nodes can harm TE, and consequently reliability; and that increasing the number of Gateways alone does not significantly worsen ContextNet performance.

8.6 Results using Scenario 2, Scenario 3 and Scenario 4

The configurations for Scenarios 2 (mtd_1), 3 (mtd_3, mtd_5) and 4 (mtd_2, mtd_4) are shown and identified in Figure 15. In all configurations 5000 MNs are connected and there is one active Gateway (GW). In 15 it is also specified whether MTD is active (MTD) or not; the period of time that each mobile node remains disconnected at each disconnection (Desc. Time); and the share (in percentage) of the MNs that repeatedly disconnect themselves from the Gateway.

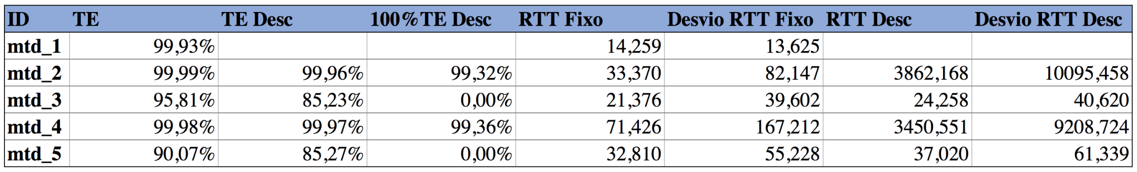


ID	NM	GW	MTD	Tempo Desc.	%MN Desc.
mtd_1	5000	1	Inativo	0	0
mtd_2	5000	1	Ativo	30000	30
mtd_3	5000	1	Inativo	30000	30
mtd_4	5000	1	Ativo	30000	70
mtd_5	5000	1	Inativo	30000	70

Figure 15: Settings for Scenarios 2, 3 and 4

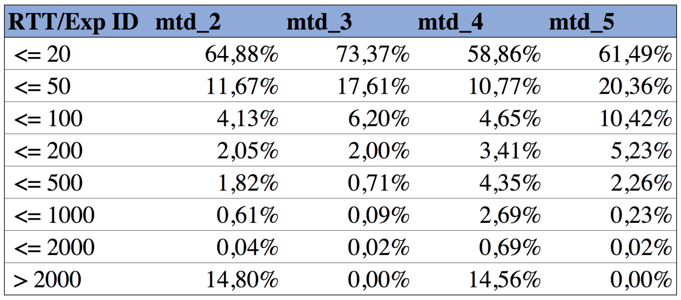
Figure 16 shows the results of the tests with Scenarios 2, 3 and 4. For each configuration, it shows the total message delivery rate (TE); the delivery rate of messages addressed to disconnecting nodes (TE Desc); the share (percentage) of disconnecting nodes whose delivery rate is 100%; the average (Fixed RTT) and standard deviation (Fixed RTT Deviation) of the round-trip-time of fixed nodes; and the average (RTT Desc) and standard deviation (RTT Desc Deviation) of the round-trip-time of disconnecting nodes.

Figure 17 shows the distribution of all RTTs for the mtd_2, mtd_3, mtd_4 and mtd_5



ID	TE	TE Desc	100%TE Desc	RTT Fixo	Desvio RTT Fixo	RTT Desc	Desvio RTT Desc
mtd_1	99,93%			14,259	13,625		
mtd_2	99,99%	99,96%	99,32%	33,370	82,147	3862,168	10095,458
mtd_3	95,81%	85,23%	0,00%	21,376	39,602	24,258	40,620
mtd_4	99,98%	99,97%	99,36%	71,426	167,212	3450,551	9208,724
mtd_5	90,07%	85,27%	0,00%	32,810	55,228	37,020	61,339

Figure 16: Results of Scenarios 2, 3 and 4



RTT/Exp ID	mtd_2	mtd_3	mtd_4	mtd_5
<= 20	64,88%	73,37%	58,86%	61,49%
<= 50	11,67%	17,61%	10,77%	20,36%
<= 100	4,13%	6,20%	4,65%	10,42%
<= 200	2,05%	2,00%	3,41%	5,23%
<= 500	1,82%	0,71%	4,35%	2,26%
<= 1000	0,61%	0,09%	2,69%	0,23%
<= 2000	0,04%	0,02%	0,69%	0,02%
> 2000	14,80%	0,00%	14,56%	0,00%

Figure 17: Comparison of Outbound Configurations

configurations in millisecond ranges. Figure 17 shows that, for example, 64.88% of the RTTs for the mtd_2 configuration were shorter than 20 milliseconds, and 14.80% were longer than 2000 milliseconds.

Comparing mtd_1 with mtd_3 and mtd_5, we evaluated the impact of network failures on RTT and TE. According to Figure 16, TE fell from 99.93% in mtd_1 to 95.81% in mtd_3; and 90.07% in mtd_5. There was also a significant increase in the RTT of even the fixed nodes when there are network failures: in mtd_1 the average was 14.25 milliseconds, in mtd_3 21.37 and in mtd_5 32.81. In other words, network failures have a negative impact even on the performance of nodes that don't fail, and reduce reliability.

Comparing mtd_1, mtd_2 and mtd_4, we evaluated the impact of MTD and network failures on RTT and TE. According to Figure 16, the TE remained practically identical in mtd_1, mtd_2 and mtd_4 and was 99.9%, i.e. MTD made communication in an environment with faults as reliable as in one without. In mtd_2 there is a significant increase in RTT when compared to mtd_1, but even with this increase the communication performance is still satisfactory. As can be seen in Figure 17, more than 58% of the messages carried on mtd_2 and mtd_4 have an RTT of less than 20 milliseconds.

Comparing the pairs of configurations mtd_2 and mtd_3; and mtd_4 and mtd_5 shows how effective MTD is; and what impact it has on RTT and TE in failure scenarios. There was an increase in the RTT Desc in mtd_2 and mtd_4, which went from tens of milliseconds in mtd_3 and mtd_5 to thousands of milliseconds; an expected behavior since in mtd_2 and mtd_4 the MNs receive "replayed" messages addressed to them at times of disconnection.

According to Figure 16 among the nodes that disconnect, the delivery rate (TE Desc) increased from approximately 85% in mtd_3 and mtd_5 to 99.96% in mtd_2 and mtd_4. In addition, in mtd_2 and mtd_4 more than 99% of the nodes that disconnected received 100% of the messages addressed to them, while in mtd_3 and mtd_5 this percentage is 0%. In other words, with MTD there was a considerable increase in communication reliability in a scenario where there are network failures/ diconnections.

Analyzing 17 we see that the RTT distribution in the ranges up to 2000 milliseconds is very similar between mtd_2, mtd_3, mtd_4, mtd_5. In the range above 2000 milliseconds there is a big difference between the configurations that have active MTD (mtd_2 and mtd_4); and those that don't (mtd_3 and mtd_5). In mtd_2 and mtd_4 more than 14% of the messages have an RTT of more than 2000 milliseconds, while in mtd_2 and mtd_4 there is no RTT of this duration. These messages whose RTT duration is over 2000 milliseconds are most likely messages resent by MTD, messages whose large delay was already expected.

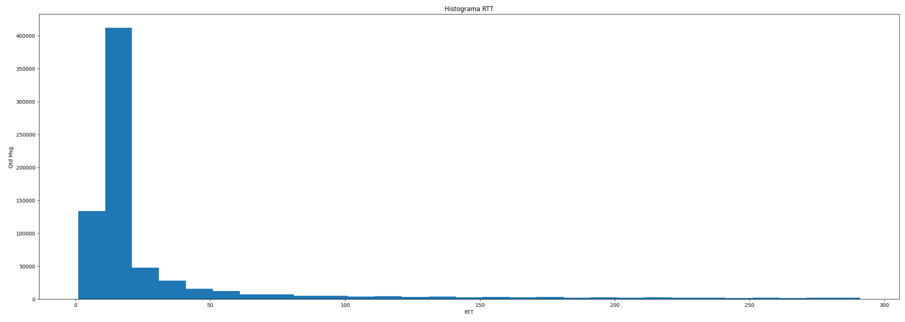


Figure 18: Histogram of Average RTT Static Nodes - Configuration mtd_4

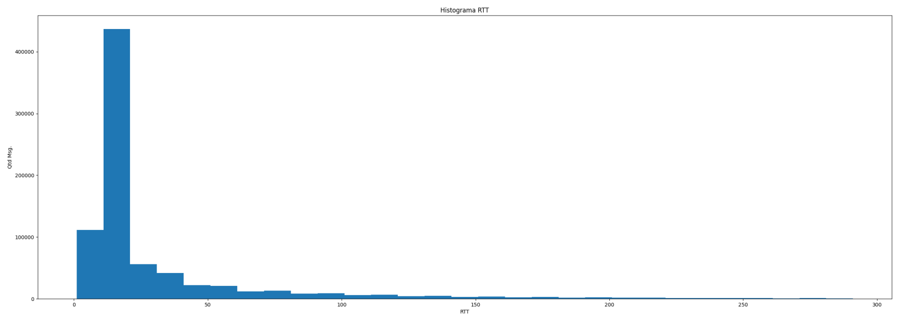


Figure 19: Histogram of Average RTT Static Nodes - Configuration mtd_5

Observing the similarity between the average RTT distributions of the stationary nodes shown in figures 18 and 19, we can say that the behavior of the Fstationary nodes is very similar with and without active MTD, despite the significant increase in the total average RTT with MTD as shown in 16.

Comparing the mtd_3 and mtd_5 configurations, we note that even with the increase in the number of nodes that disconnect, the TE Desc remains practically constant as shown in 16. The same can be observed when comparing the mtd_2 and mtd_4 configurations, which leads us to conclude that the number of nodes that disconnect has little influence on TE Desc.

8.7 Results of outbound communication for Scenario 1 and Scenario 2

Now we will evaluate the performance of outbound communication (Scenario 2, configuration mtd_1) and compare it with Scenarios 1 and 2.

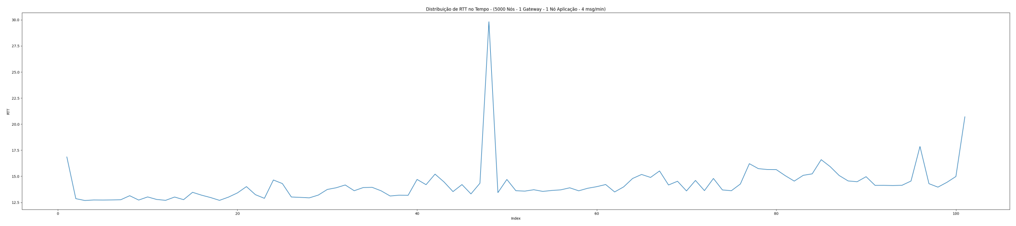
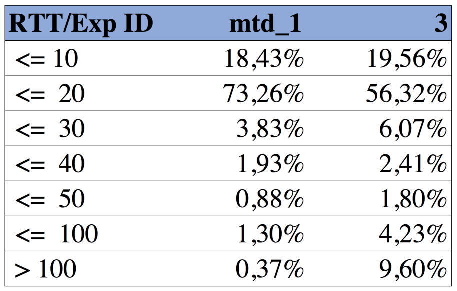


Figure 20: Time Distribution - Configuration mtd_1

To check that the performance of ContextNet Kafka Core remains constant for outbound communication over time, Figures 20 was plotted. It plots the average RTT per sending order. The order varies from 0 to 100, as 100 messages were sent per node. Looking at Figures 20, it can be seen that the performance of ContextNet outbound communication remains almost constant over time.

From 16 we know that TE for mtd_1 is greater than 99.9%, i.e. ContextNet's outboud communication is reliable. From 16 we also know that the RTT for mtd_1 is 14.25 milliseconds, and the standard deviation is 13.62. We can therefore conclude that the performance of the new ContextNet for Outbound communication is satisfactory.



RTT/Exp ID	mtd_1	3
<= 10	18,43%	19,56%
<= 20	73,26%	56,32%
<= 30	3,83%	6,07%
<= 40	1,93%	2,41%
<= 50	0,88%	1,80%
<= 100	1,30%	4,23%
> 100	0,37%	9,60%

Figure 21: Comparison of the Configurations mtd_1 and mtd_3

To compare the performance of Inbound and Outbound communications, we selected configuration 3 from Scenario 1 and mtd_1 from Scenario 2. They have the same number of connected MNs; both are being run by a single node; and the frequency of messages sent is the same in both.

According to Figure 4, the average RTT of configuration 3 is 49.45 milliseconds, significantly higher than that of configuration mtd_1, which is 14.25 milliseconds according to Figure 16. Looking at the histograms in Figures 22 and 23, which show the distribution of the average RTT of each node, we can see that in mtd_1 there are many more nodes whose average RTT is less than 12.5 milliseconds, which is not the case in configuration 3. Figure 21 shows that more than 91% of the messages in mtd_1 have an RTT of less than 20 milliseconds, while in configuration 3 this percentage drops to 75%. Thus, there are indications that Outbound communication performs better than Inbound, but more configurations should be tested to corroborate this hypothesis.

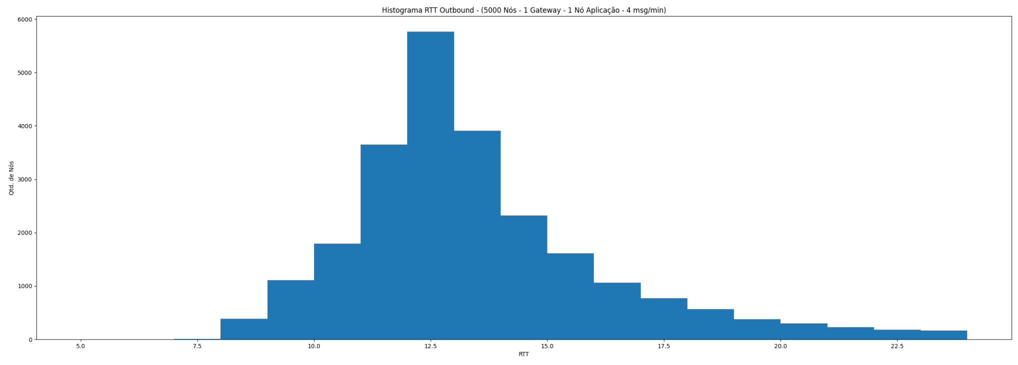


Figure 22: Histogram of Configuration mtd_1

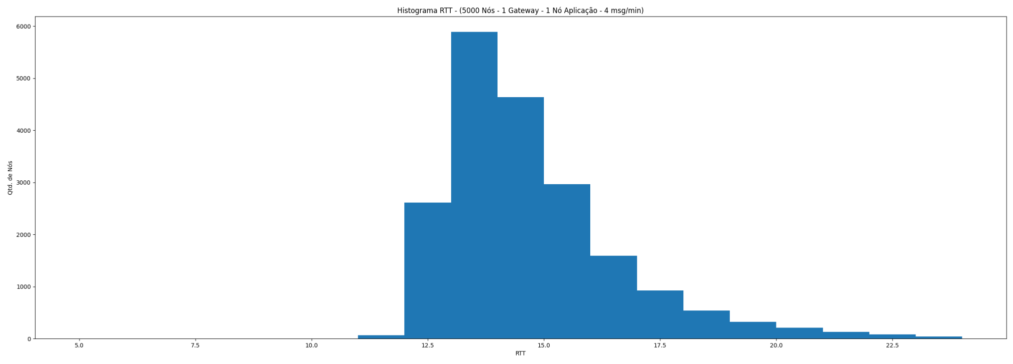


Figure 23: Histogram of Configuration 3

8.8 Discussion

In this section will discuss factors that influenced the execution of the tests but which are not being evaluated by them, or which are beyond the scope of this work.

As mentioned in the previous subsections, MR-UDP is a protocol that consumes a lot of memory from the Java virtual machine at the time of the first connection. To prevent the Java virtual machine from failing, an interval was established between connections. This interval ensures that the *Garbage Collector* can act and increase the memory available for the next connection, however, it slows down the execution of the experiments. All the types of *Garbage Collector* that the Java virtual machine natively has at its disposal were tested, and after the experiments it was concluded that the *Garbage First Garbage Collector* [14] was the most appropriate for use with the Gateway, and therefore with MR-UDP, as it allowed the interval between connections to be shorter than the others. In addition to the type of *Garbage Collector*, there are various configurations of the Java virtual machine that can influence the performance of applications, but which were not evaluated in this work.

During the experiments, it was also observed that the MR-UDP protocol would break connections if they were left without messages for a few minutes. This behavior of the protocol required that in all scenarios during the process of connecting the mobile nodes to the Gateway (before the experiment *per se* began), Core to MN, and MN to Core messages were sent to or by all MNs at intervals of less than one minute.

Both the applications and the Gateway used *Thread Pools* [15] to manage the parallel *Threads*. Before running the test scenarios, experiments were carried out varying the size of the *Thread Pools* of the Gateways and applications and measuring the RTT. These experiments were carried out in order to obtain an effective size, i.e. a size that would make the RTT satisfactory.

As expected, it was observed that the RTT is dependent on size of the *Thread Pool*. It was also observed that the effective size depends on the number of mobile nodes in the experiment. With these experiments, an effective size was found for the Gateway - the same for all tests - and for each application node. There is an ideal size, one that would make the RTT minimal, for each *Thread Pool* of the tests described above, but it is beyond the scope of this work to find it.

9 Conclusion and Future Work

As IoT becomes more and more popular both in industry and in people's daily lives, the challenges facing the applications that support it increase. In this context, this work describes a new version of the ContextNet Core *middleware* whose function is to support scalability, parallelization and high availability of the back-end services of these applications executing in a stationary network (cluster or a cloud). In this new version of the ContextNet Core we used Kafka as the underlying communication infrastructure.

The tests carried out in this work indicate that communication in the new ContextNet Kafka Core is *real-time*. In addition, it was shown that it supports large numbers of mobile nodes and allows parallelization of IoT applications. In this study, we learned that the parallelization of some IoT applications may be a key factor in their viability and that there are indications that the parallelization of applications improves execution performance, such as lower delays.

The experiments also showed that some of the added functionalities, such as the temporary storage of messages addressed to disconnected mobile nodes, performed as expected when they were conceived.

The aim of making it easier for developers to use ContextNet Kafka Core was also achieved, since the image of all the components presented are in docker published and available for use.

The test results showed that there is no significant difference in terms of speed between the former and new ContextNet *intra-Core* communication. Both DDS and Kafka performed well; however, Kafka has a number of features that make it suitable for applications whose needs go beyond the basics, as well as simplifying implementation for the developer.

The principles of ContextNet applied to Kafka have enabled the construction of a *middleware* that meets the needs of mobile communication and IoT applications, and not just a fast communication bus.

9.1 Future WorK

There are a number of test scenarios not explored in this work that could be carried out in order to evaluate the new features and performance of the new ContextNet Core even more thoroughly. For future work, we intend to analyze scenarios with and without the PoA-Manager; repeat the scenarios already evaluated with a greater number of variables in order to gauge the impact of each one more precisely; test applications with significant processing needs and scenarios with simultaneous *Inbound* and *Outbound* communication.

It is also part of future work to document in detail how to use the components of ContextNet Kafka Core; how to build and parallelize applications; and how to migrate applications developed from the old ContextNet to the new ContextNet.

References

[1] ENDLER, M.; E SILVA, F. S. Past, present and future of the contextnet iomt middleware. *Open Journal of Internet Of Things (OJIOT)*, v. 4, n. 1, p. 7–23, 2018.

[2] DDS What is DDS? `https://www.dds-foundation.org/what-is-dds-3/`, 2021.

[3] OMG. Data Distribution Service for Real-time Systems Specifications. www.omg.org/spec/, 2012. (visited on Sept. 28, 2012).

[4] WANG, N.; SCHMIDT, D. C.; VANT HAG, H.; CORSARO, A. Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (ncow) systems. In: MILCOM 2008 - 2008 IEEE Military Communications Conference. c2008. p. 1–7.

[5] DAVID, L.; VASCONCELOS, R.; ALVES, L.; R., A.; M., E. A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes. *Journal of Internet Services and Applications*, v. 4, n. 16, 2013.

[6] ENDLER, M.; BAPTISTA, G.; SILVA, L. D.; VASCONCELOS, R.; MALCHER, M.; PANTOJA, V.; PINHEIRO, V.; VITERBO, J. ContextNet: Context Reasoning and Sharing Middleware for Large-scale Pervasive Collaboration and Social Networking.

In: Proceedings of Middleware Conference - Posters and Demos Track. c2011. p. 2:1–2:2.

[7] Kafka 2.6 Documentation. https://kafka.apache.org/documentation/, 2020. Last accessed September 2020.

[8] WAEHNER, K. Event streaming and apache kafka in telco industry. https://www.kai-waehner.de/blog/2020/03/06/event-streaming-apache-kafka-telecommunications-industry-telco-business/, 2020. Last accessed August 2020.

[9] DíAZ, M.; MARTíN, C.; RUBIO, B. -CoAP: An Internet of Things and Cloud Computing Integration Based on the Lambda Architecture and CoAP. In: 11th Collaborative Computing: Networking, Applications, and Worksharing. c2015. p. 195–206.

[10] RANJAN, Y.; KERZ, M.; RASHID, Z.; BöTTCHER, S.; DOBSON, R.; FOLARIN, A. Poster: Radar-base: A novel open source m-health platform. In: . c2018. p. 223–226.

[11] SNEPPE, M.; NAMIOT, D. On mobile cloud for smart city applications. In: . c2016.

[12] SILVA, L. D. N.; ENDLER, M.; RORIZ JR., M. MR-UDP: Yet Another Reliable UserDatagram Protocol, now for Mobile Nodes. Technical report, Departamento de Informática, PUC-Rio, 2013. Monografias em Ciencia da Computação, MCC 06/2013.

[13] VASCONCELOS, R.; SILVA, L.; ENDLER, M. Towards efficient group management and communication for large-scale mobile applications. In: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on. c2014.

[14] Garbage first garbage collector tuning. https://www.oracle.com/technical-resources/articles/java/g1gc.html.

[15] Thread pool executor. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html.