University *POLITEHNICA* of Bucharest
Faculty of Automatic Control and Computers, Computer Science and Engineering Department

PhD Thesis Summary

in Computer Science, Information Technology and System Engineering

Distributed Object Oriented Runtime System

Sistem distribuit orientat pe obiecte

presented by

Drd.ing. Dorin PALANCIUC MAWAS

supervised by

Prof.dr.ing. Florin POP

2021
Bucharest, Romania

Contents

1 Introduction **2**
1.1 Main features . . . 2
1.2 Structure of research . . . 3
1.3 Thesis outline . . . 3

2 Proposed Solution **5**
2.1 Object and event orientation . . . 5
2.2 Distributed architecture . . . 6
2.3 The target problems . . . 6
2.4 Orientation towards edge computing . . . 7
2.5 A minimal system and first measurements . . . 7
2.6 Overall architecture . . . 8
2.7 Nodes and interfaces . . . 9
2.8 Inter-node communication . . . 10

3 System Level Capabilities **11**
3.1 Notifications . . . 11
3.2 Service requests . . . 12
3.3 Detecting faults . . . 13
3.4 Replication . . . 14
3.5 Partitioning . . . 16
3.6 Routing of messages . . . 18
3.7 Achievable consistency models. Transactions . . . 19
3.8 Concurrency control . . . 21
3.9 How eventual is eventual consistency? . . . 22

4 Node Level Capabilities **23**
4.1 Node communication . . . 23
4.2 Objects in DOORS . . . 23
4.2.1 Defining structure. The DOORS type system . . . 24
4.3 DOORS system services . . . 25
4.4 Administration Services . . . 26
4.5 Introspection . . . 26
4.6 Object Persistence, tracking and threading . . . 26
4.7 Concurrency control at node level . . . 27

5 Application Level Challenges **28**
5.1 Deployment and versioning of behaviour and structure . . . 28
5.2 Modelling of events and their processing . . . 29

5.3 Application-level security 30

6 Case Studies **32**
6.1 Case study - Telemetry 32
6.2 Case study - Edge Computing 34

7 Conclusion **36**
7.1 Summary of contributions 36
7.2 List of publications 36
7.3 Future work 37

References **38**

1 | Introduction

We aim to define a simple and efficient way of building distributed systems, starting from a minimal set of features needed, and keeping the design as simple as possible, since much of the complexity in distributed systems currently operated is accidental, emerging from the use of multiple paradigms, technologies and approaches.

Our solution is relevant when the sources of data are scattered over a territory, or when the place where a decision needs to be taken in a system is far from the place where resulting actions need to be performed (e.g. SCADA, telemetry and IoT) . DOORS is also a viable alternative when using many machines together for achieving economy of scale, or resilience through redundancy.

DOORS promotes edge computing. Data storage and computation are brought closer to the sources of data. Immediate benefits would be the significant decrease of communication bandwidth needs, and the real opportunity to decrease communication latency. In addition, as mobile devices become increasingly powerful and numerous, more elaborated constructs emerge, fostering the same decentralisation tendency, such as *Drop Computing* [8], on which carefully designed scheduling solutions promise to fulfil even real-time constraints [29].

1.1 Main features

The proposed design is a runtime environment. The system implementer would only need to install the nodes at their desired locations, interconnect them and then execute the runtime on each. The actual solution to the distributed problem needs then to be described in terms of both structure and behaviour as a set of objects, hosted and ran on the instances of the runtime which are operating on each node. These app-specific objects shall have access to communication, consensus, replication and transactional support primitives, provided as services by the runtime.

The environment provides a low level of abstraction: single-threaded objects, holding state, sharing nothing and exchanging messages. DOORS provides no shared-memory abstraction, or garbage collection. DOORS focuses on offering support for the following instead:

- deployment and evolution during normal operation;
- management of message streams(in fact event streams);
- measuring system performance in real time;
- migration of objects from one node to another;
- reaching consensus and transactional updates of data.

1.2 Structure of research

Starting with the characteristics of the targeted problems: geographical distribution and weak communication channels, we enumerate the desirable set of traits: providing an abstraction encapsulating both state and behaviour - the *object*. We "endow" objects with the abilities to exchange asynchronous messages, to evolve during operation, to move from one node to another during operation, and to decide this based on out-of-the box real-time performance metrics.

The research is guided by the following questions:

1. **What is the minimal set of primitive mechanisms we need to build into our design in order to be able to provide the desirable set of traits enumerated above? (RQ1)**

2. **What are the system-level capabilities we need to provide in order to be able to resolve the targeted class of distributed problems? (RQ2)**

3. **What are the node-level capabilities which need to be present and how are they being provided by the design? (RQ3)**

4. **What are the challenges emerging at application level (taking into consideration the distributed nature of the applications) and how does DOORS address them? (RQ4)**

5. **How is DOORS achieving its consistency and availability goals by implementing partitioning and replication? (RQ5)**

Our objectives map on the research questions,and are listed in Table 1.1.

1.3 Thesis outline

This thesis focuses on the system design, listing the set of required functionality and then describing how the system fulfils them. Where the case, experiments were performed, each of them developed on the "reference implementation". Beginning with the present introductory chapter, the thesis continues as follows:

Chapter 2 gives an outline of the solution, presenting the major components and allowing to later place details on each.

In the **Chapter 3** describes the set of basic distributive capabilities. These include the nature and types of messages exchanged, with their distinct objectives (i.e. notifications, versus services calls and the detection of system-level faults), and the two most important capabilities present at system level, which are *replication* and *partitioning*.

In the next section, **Chapter** 4 we "descend" at node level. All the above-described system capabilities rely on mechanisms and features present on each node. We start with the ability of inter-node communication, describing the design choice and continue with the abstraction provided by DOORS, which is the *object*. DOORS objects are different from the traditional OOP approach by executing methods asynchronously as response to message receipts, by their single-threaded nature, as well as by how their life-cycle is managed by the system (no garbage collection).

Chapter 5 presents the challenges which appear at application level and how they are addressed by the design described in the previous chapters. The topics covered here are: security, application deployment, versioning of structure and behaviour, and how we model and process events.

Research question	**Research objective**	**Research methodology**	**Use case**
RQ1	Establishment of the foundation of the system, taking into consideration the set of traits considered desirable for efficiently solving the targeted problems	Prototyping and evaluation	Identification of alternative architectural solutions. Refinement into reusable components
RQ2	Functional definition of the system. Outline of the structure present at global level	Incremental development of the design	Asynchronous communication. Receipt confirmation. Matching of requests and responses. Replication of object state. Partitioning of objects across the available set of nodes. Transactional support. Validation against the targeted set of system traits
RQ3	Functional definition of the node. Outline of the structure and mechanisms present at local level	Incremental development of the design	Description of object structure and behaviour. Scheduling of object execution. Concurrency control
RQ4	Identification of the most common challenges encountered in deployment, operation and maintenance of the distributed systems implemented for solving the targeted problems	Qualitative approach, based on the analysis of publications and documentation	Updates during normal operation. Application level security. Modelling of events, sourcing and complex event processing
RQ5	Identification and description of the relevant consistency models	Qualitative approach, based on analysis of academic publication and documentation of industry-standard solutions	Object updates during various integrity scenarios: complete system, isolated node, segmented system. Deduction of the achievable consistency model in each scenario

Table 1.1. Thesis objectives and methodology.

Chapter 6 contains case studies. These are concrete implementation examples, where we outline how the features of the system solve the specific challenges.

The thesis ends with **Chapter 7**, which sums up the contributions, outlines the "best-case scenario" industrial implications of the DOORS proposal, and a brief description of the future work envisioned on the implementation of DOORS.

2 | Proposed Solution

We propose a set of principles, centred around the following features: object orientation, event orientation, and distributed architecture.

2.1 Object and event orientation

The unit of modelling is the object, mapping to an entity from the problem space. The object abstraction shall encapsulate both the state and the behaviour of the modelled entity or concept. Objects communicate by sending each other asynchronous messages. The receiver "decides" how to respond to each message.

Messages are used to model events. The system reacts to events received from its environment and the objects composing the system define their behaviour in terms of reactions to events. The system shall also provide support for acquisition and processing of event streams, which are virtually infinite series of events. Objects are "instances" of classes. Classes and objects are persistent, which relies on the fact that DOORS nodes run on systems with continuous power. The system runtime offers introspection into all hosted objects. Even the system objects shall present their interfaces in the same manner. A properly authorised client shall be able to explore and extend the structure of the system at runtime.

No ORM (Object-Relational Mapping) and/or relational database systems are employed in order to provide persistent state for objects. Objects may refer to other objects, so a class definition may describe its components in terms of other classes. The system provides no garbage collection. Unless explicitly deleted, an object continues to exist indefinitely in the system. Objects that are not "used" for a long time shall end up in storage, as shown in Figure 2.1:

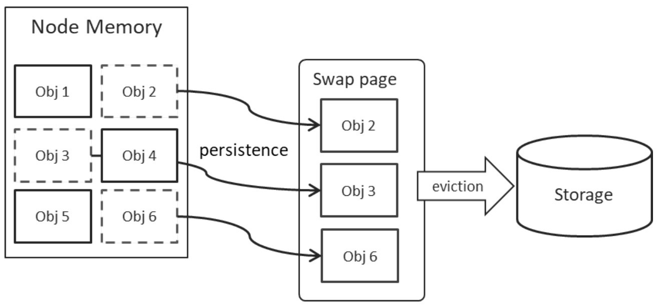

Figure 2.1. Object persistence and eviction to storage.

In the reference implementation there is no inheritance, or polymorphism.

2.2 Distributed architecture

The system is composed of a set of units with computation and storage capabilities, named "nodes". Different nodes may have radically different specifications and may be distributed geographically, but they are all interconnected. DOORS provides moderate scaling needs, in the range of tens to hundreds of nodes per system instance. Nodes provide for both replication and partitioning. The system provides services to clients, which also use network connections. A client may connect to any of the nodes and may send messages to any object in the system, regardless of its node of residence. The state of any single object resides on a single node, while its behaviour specification is replicated on each node, enabling parallel execution and scalable storage.

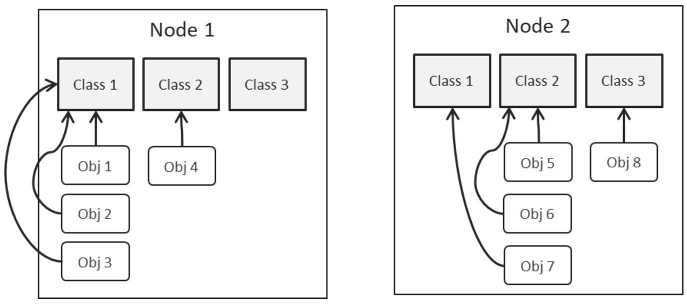

Figure 2.2. Replicated classes, partitioned objects in a two-node DOORS instance.

Figure 2.2 shows an example of a two-node DOORS instance, containing 3 classes and 8 objects The arrows depict the fact that each object refers its corresponding class definition.

Objects may be grouped in collections, which in turn may be spread across the available nodes. The runtime shall provide services allowing for objects to "migrate" themselves between nodes.

Failures are assumed to be non-byzantine and total. A failed node becomes totally inaccessible, and its peers would not be able to determine whether the failure is located in the node or in its network connection. The system shall detect failures and provide a configurable consistency model (ranging from *strict serializable* to *read uncommitted*).

Minimum node turnover is expected. The system is not meant to be one in which a great number of peers join and leave frequently.

2.3 The target problems

We adjusted the focus of DOORS to the sub-set of distributed problems with the following characteristics:

- many, heterogeneous, geographically distributed data sources and data sinks (e.g. telemetry devices, control equipment);
- the data sources have limited local resources. The communication channels may be unreliable, have low throughput, or high latency;
- the data contained needs to be processed and aggregated into central repositories;
- some of the processing and outputs need to be elaborated with low latency, mandating for processing ”in the field”;
- the system is large enough or mission-critical, such that functional updates need to be implemented during ”production run-time”.

Figure 2.3 depicts an example, with remote nodes, specialised in data acquisition or

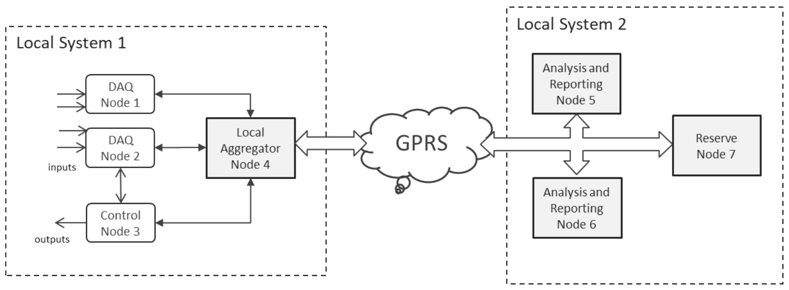

Figure 2.3. Example of geographically distributed DOORS installation.

control, and central nodes, specialised in data aggregation and analytics. The system contains local control loops, a centrally located cluster of computing capacity and unreliable network connections towards remote nodes.

2.4 Orientation towards edge computing

Edge computing, as well as *fog computing* are closer to the goals of DOORS, due to their orientation towards decentralization.

We acknowledge the approach of *Drop computing*, which is oriented towards ad-hoc collaboration, and aims for higher levels of scalability [26].

2.5 A minimal system and first measurements

When DOORS development was at the beginning, we outlined and developed its basic architectural and functional features. We started with a basic system, able to host objects of a single class, containing two attributes with integer values, *a* and *b*, and two methods, one with side-effects and the other one without.

In this context, the first operation performed by a generic DOORS application is creation of objects. This involves the actual allocation and initialisation of the object, as well as the correct initialisation of system structures. Our first experiments test the capacity and scaling capabilities of the set of data structures chosen to store the Object Dictionary. We considered two alternative test platforms: a virtual machine running on Virtual Box, using two CPU cores running at 2.5 GHz and 8GB of RAM, and a physical machine running on a dual-socket machine, running at 3.06 GHz and 32GB of RAM. Both machines run the same version of Linux, Manjaro 21.1.2, with kernel version 5.10.

The physical machine had a clear performance advantage (rather similar results in case of a small number of objects, but approximately 6 times faster in case of larger number), and a higher consistency in results across the entire set of test rounds. The standard deviation achieved on the virtual platform for the 64k set was 4900, two orders of magnitude larger than the one achieved on the physical platform, which was 43.1.

The results recorded on the physical machine are depicted in Figure 2.4.

The final value is extrapolated, based on the recorded series. A duration of 72 seconds is needed for creating a dictionary of 64000 objects, meaning we need almost 9 minutes for creating a dictionary of one million objects. Creating the dictionary "from scratch" does not scale satisfactorily, and DOORS must build the dictionary incrementally and store it, in order to be usable.

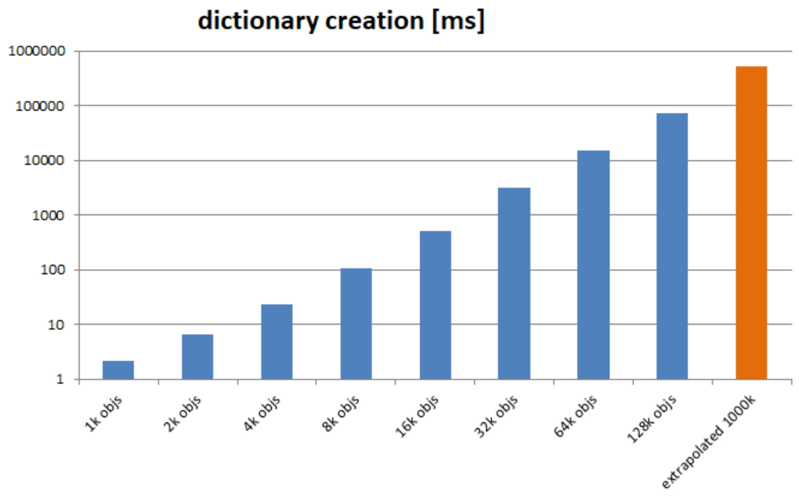

Figure 2.4. Object Dictionary creation time, logarithmic scale.

2.6 Overall architecture

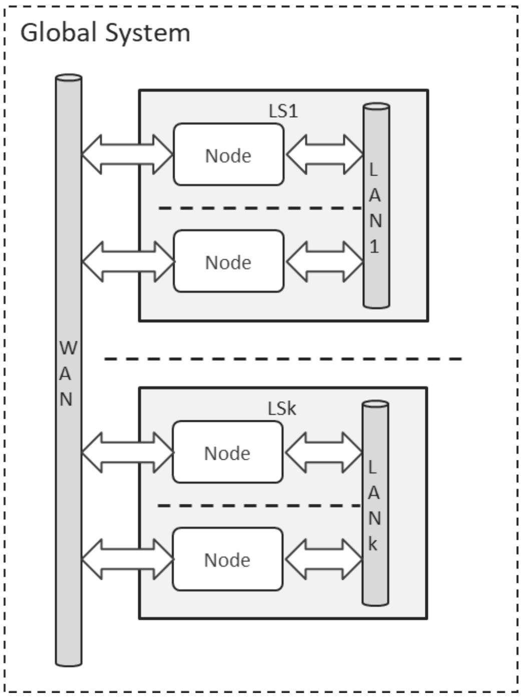

Figure 2.5. System architecture on two levels.

- Local System - several nodes sharing the same local network, with low latency and a high level of trust;
- Global System - several Local Systems grouped in the same DOORS instance, across high latency, mainly non-trusted, network connections (i.e. Internet).

A set of k Local Systems is depicted in Figure 2.5, within a single GS instance. The local networks are labelled ”LAN 1” through ”LAN k” each accessible only from their respective LS, while the non-trusted, high latency WAN is accessible to each node.

2.7 Nodes and interfaces

We shall identify nodes by their ID, which are positive integers, assigned statically. Each DOORS node exhibits 3 types of interfaces, based on their purpose: client, node and administration interfaces.

The Client interface provides DOORS services to external clients. Examples of services: creation, updates, and deletions of class definitions, instantiation of classes (creation of objects), selection of objects, sending messages to objects and receiving responses, deleting objects. A DOORS client should connect to only one node and issue all service requests to the respective node. If the request targets an object residing on a different node, then it shall be forwarded ”transparently” by the directly contacted node.

The Inter-node interface connects a DOORS node to all its peers in the same system. Each node maintains exactly one connection per peer. The relationship is not client-server, this interface is fully bidirectional and asynchronous. In a system containing N peers, node X will await connection requests for all peers with id smaller than X and initiate connections to peers with ids greater than X.

The Admin interface allows an external entity to invoke administrative operations, and enables privileged access to the system objects: joining/leaving a system, changing/reloading of node configuration, shut-down/restart, visualisation and configuration of system-level metrics.

Each node needs to have an up to date status information on its peers. When faults are detected (such as loss of connection with one of the peers), the system enters the recovery state and uses object replicas in order to maintain normal operation. We only consider non-byzantine, hard-stop types of failures. We define the following possible node states, with regard to its connectivity with "the rest of the system": OFF-LINE - The node is not accessible from the system, UNSYNC - The node has just became accessible, RECOVERY - The node is in progress of synchronising with the rest of the system, therefore not yet operational, and SYNC - The node has successfully synchronised with the rest of the system and is fully online and operational.

During its lifetime, a node progresses through the above-defined states in increasing order, up to the SYNC state. In case a new loss of connection occurs, the node state shall be recorded everywhere as OFF-LINE, from where it shall resume its progression after turning back on-line.

Each node keeps track of the state of all its peers,by a periodic message exchange, so that both channel and node failures are being identified quickly (”heartbeat” protocol). According to measurements performed in a non-perturbed 780Mbps wi-fi network, the duration of one status inquiry is around 10ms, and involves the transmission of less than 60 bytes (it involves three one-way trips between the nodes). Given the fact that one message exchange yields status for both involved nodes, the number of messages needed for a full status update of an N-node system is N*(N-1)/2. The total duration of the status update as a function of the number of peers is depicted in Figure 2.6:

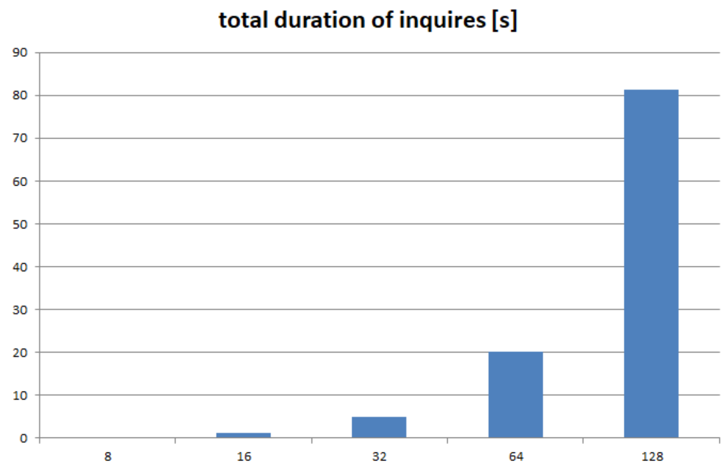

Figure 2.6. Evolution of the total duration of status requests as a function of the number of peers in the LS.

A DOORS system of 128 nodes needs 90 seconds to get a full status update. If we restrict the status inquiry traffic to a maximum of 25% of traffic, we would never be able to get a status update more often than any 325 seconds. Network congestion is avoided if the status inquiries are only exchanged in the absence of useful traffic.

2.8 Inter-node communication

We propose a layered approach. DOORS nodes exchange byte streams over TCP sockets, structured into messages of various (but known) lengths. Messages exchanged on all node interfaces have the same, straightforward structure, containing a start byte, a length field and the payload. Even though the completeness and integrity of the transmitted data is guaranteed by the underlying TCP, we fragment the traffic into messages of known size so that we can catch timeouts as soon as possible. The implementation will detect timeout exceptions, as thrown by the communication library, and trigger the needed status changes in the runtime. As the message length is encoded on two bytes, we cannot transmit messages longer than 64 kB, but sizes as low as 16kB have proven adequate.

On top of the messaging framework we define two types of messages: notifications and service requests. Both types shall receive an "immediate" answer (best effort). In the case of notifications, the answer is just a confirmation receipt, as described in Section 3.1. In case of service requests, described in Section 3.2, the answer is composite, containing: a status - success or failure of the invocation, and further data – in case of success, there will be means to access the result (i.e. ID of objects or even serialized objects). in case of failure, there will be error messages, line number references, so on.

As services are executed asynchronously, the answer is available later (e.g. longer than the node status latency timeout), so the system enables the requester to pair-up the answer with the corresponding request, as described in Section 3.2.

3 | System Level Capabilities

We are presenting the DOORS approach in providing resilience and scalability while maintaining the highest achievable levels of consistency and availability in the presence of faults. Even when the physical means of communication between the nodes are uniform (i.e. everything runs on asynchronous messages sent over TCP sockets), there are several "kinds" of communication:

- notification: an object on node A informs another object on node B about an event which occurred, and does not expect any response;
- service call: an object on node A invokes a service exposed by another object on node B and expects an answer at some point in time;
- replication: the system copies an object residing on node A on a potentially nearby node B;
- migration: the system moves an object residing on node A on another node B in order to optimise the use of available resources.

3.1 Notifications

Notifications are messages which do not expect and answer and describe a business-relevant fact or event. DOORS implements explicit and immediate confirmation of notifications, with responses, so that it can identify and address communication faults. Depending on the exact moment of fault occurrence, there might be no confirmation, exactly one confirmation, or multiple identical confirmations received.

This is depicted in Figure 3.1 below :

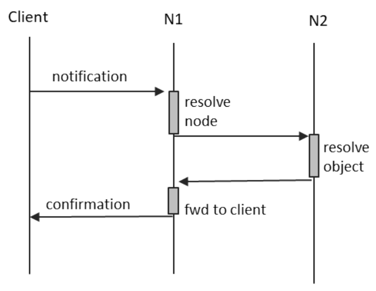

Figure 3.1. Notifications.

A connection fault between the client and N1 means that the notification will not reach N1 and no confirmation is ever received. A fault on the path between N1 and N2 means that the notification will not reach its destination, and again no confirmation is available. We have two possible alternatives:

- N1 may store the notification locally, as "offline message" and enqueue it for sending it later, when the connection to N2 is restored. A conservation period may be defined, after which N1 shall discard the cached notification;

- N1 may detect the loss of connection to N2, and send back to the client an error message. This case is depicted in Figure 3.2.

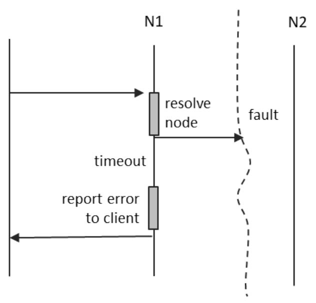

Figure 3.2. Notification fault.

DOORS implements idempotency: even when a node receives the same message twice, it is able to ascertain that the two messages refer to the same event. Both nodes compute a quasi-unique ID of any message - a simple digest. This ID is used then to determine if a certain message is received multiple times.

3.2 Service requests

Node A is sending a message to node B and expects an answer in return. As DOORS has asynchronous behaviour, the response of node B may arrive at a much later moment in time. We have to address the following:

- pairing up a request with a much later answer (potentially received after several other requests, maybe even "out-of-order");
- deciding when to "stop waiting" for an answer;
- determining whether a request produced any effects on the destination object;
- de-duplicating multiple responses.

The unique identification of the request (by the receiving node) and the inclusion of the ID into the response is used for de-duplication and pairing requests with responses. For the timing related challenges we rely on the conclusion of Fischer and Linch in [16], that on fully asynchronous systems, consensus is impossible to obtain in the presence of node crashes. DOORS limits the period in which an object must reply. If this interval expires, both the sender and the receiver assume that the service request failed and act accordingly (either abandon or re-attempt the request, cancel the service execution and roll back any partial changes already performed).

DOORS assimilates service calls to objects invoking methods of other objects. In addition to the normal confirmation of receipt message, the receiver sends back a second response, containing the expected result. The ID constructed by the sender shall be used in a manner similar to the notification case, to pair up the initial request, the receipt confirmation as well as with the final response, and also to de-duplicate answers. The process is depicted in Figure 3.3.

The asynchronous semantics for service calls in DOORS mean that:

- if no receipt confirmation is received in the normal message exchange time window, the client shall assume that the request produced no change of state into the system;
- if the receipt confirmation is received but contains the "DESTINATION OFF-LINE" flag, the client shall assume that the system will re-attempt transmission and execution of service request, provided that the service window has not expired;
- if the receipt confirmation is received without the "DESTINATION OFF-LINE" flag

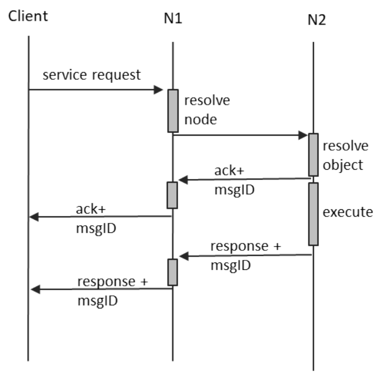

Figure 3.3. Service request scenario.

set, the client shall assume that the system successfully routed the request to the intended object and that the service is executing or shall be executed in the future, during the service window period;

- if the receipt confirmation is received without the "DESTINATION OFF-LINE" flag set, but no response is received during the service window period, the client shall assume that the service execution failed and that no side effects occurred;
- if the receipt confirmation is received and the final answer is also received, the client shall assume that the service executed and that any potential side-effects are durable.

If a fault occurs between the client and N1, or between N1 and N2 before the receipt confirmation is returned by N2, the system behaviour is similar with the case of notifications.If the fault between N1 and N2 occurs after receipt confirmation, but before the transmission of the final service response, N1 performs no further interpretation and does not create any supplementary message. This is depicted in Figure 3.4, showing that the client ends up not receiving the final service response.

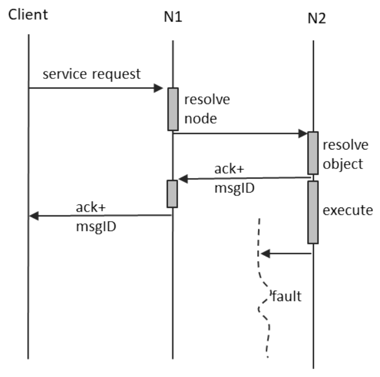

Figure 3.4. Fault during service request.

3.3 Detecting faults

Communication failure may be detected by the system (sockets and communication library), but there are also types of failures impossible to detect at socket level, e.g. a severed Ethernet cable, or a "frozen" peer node, for which we implement a periodic peer status enquiry mechanism. This ensures an upper boundary on the time elapsed until the system is aware of communication faults.

Based on the number of failed communication channels detected, DOORS distinguishes between *node isolation*, when a node lost all its channels, and *system segmentation*, in which only some of the peer connections have failed. When the integrity is restored, a potentially large set of "stale" messages have to be processed.

Besides communication failures, we might also register faults in data access. We consider the following:

- read-write conflicts - (or *unrepeatable reads*), occur when two successive reads of the same data object yield different results for one client, due to the fact that another client performed a write operation on the object in-between the two reads.
- write-read conflicts - (or *dirty reads*), occur when the read operation of one client returns a value which is going to be "immediately" modified by another client;
- write-write conflicts - (or *lost updates*), occur when the value written in an object by one client is being overwritten by another client.

3.4 Replication

Replication is the act of keeping multiple copies of the same data in distinct locations. In DOORS, replication offers protection against node and network failures.

Single-leader replication is the simplest leader-based form. It can also be called master-slave or active/passive replication and is based on the fact that only one of the replica plays the role of "leader" (or "primary" or "master"). Clients must send the write requests to the leader, which is then in charge with first executing it locally and then propagate it to the other replicas, named "followers", "secondary replicas" or "slaves". The following must be addressed:

- initial setup (choose the leader and the set of follower nodes for each object) - the optimal approach is to ensure uniform distribution of object replicas on the available set of nodes;
- detection of leader failure and election of a new leader;
- detection of follower failure and set-up of a new follower;
- synchronisation of the newly-joined follower node - objects hosted on the newly "recruited" follower node must be brought to the same state as the master replica.

Multi-leader replication ensures the ability to write on more than one replica at a time. Complexity rises significantly, as each master must act as a follower to the other masters, and faults are significantly more difficult to address.

In leader-less replication, any of the replicas accepts writes, and the first product to use approach is Dynamo [11]. Specific mechanisms, such as *Read Repair*, *Anti-entropy*, or *Quorums* need to be put in place in order to ensure propagation of changes to all replicas. Also, the read results must pass through quorum. There are profound differences in design imposed by the leader-less nature, when compared with single-master, therefore we consider leader-less replication to be out of scope for DOORS.

The ordering of events is essential in the correct resolution of write conflicts, unless we find ways making the ordering of writes irrelevant for achieving correctness. Conflict-free Replicated Data Types (CRDTs) allow this. An old example of CRDT are *Vector Clocks*, as defined by [22]. The first formal definition of CRDTs is recorded in [33]. They emerged in concurrent editing, and were proposed as alternative to Operational Transformations [13]. The solution proposed by CRDT is “Strong Eventual Consistency”, which makes sure that conflicts are merged automatically and the resulting value is correct and consistent, albeit at a later point in time. The main benefit is the ability to merge CRDTs, and this operation, receiving two CRDTs as input and is commutative, associative, and idempotent.

These characteristics are mandatory, as the global ordering of events is very difficult to determine, if at all, and to address the fact that some messages may be received more than once.

There are two major types of CRDTs: “state-based CRDTs”, also called “convergent replicated data types”and “operation-based CRDTs”, also called “commutative replicated data types”. Most of the concrete examples of CRDTs can be found in [34]. We also mention *Sequence CRDTs.* These are ordered sets, or sequences or ordered lists, and can be used to build collaborative online editors. There are quite a few documented sequence CRDT implementations: Treedoc [24], RGA [31], Woot [30], Logoot [38], LSEQ [28].

We choose DOORS to implement a single-leader replication variant, the so-called semi-synchronous replication, depicted in Figure 3.5. Only the first replica is synchronous, regardless of the replication factor, which is configurable, but unique system-wide. A single replica is master and shall be used for reads, writes, and for executing methods.

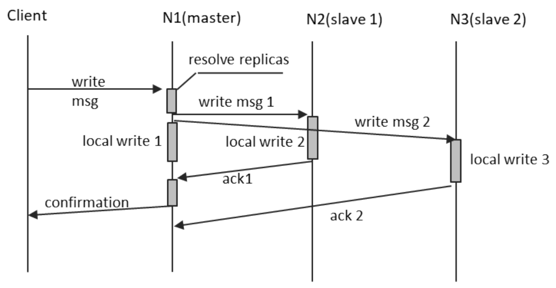

Figure 3.5. Semi-synchronous replication.

In case of node failure, the system enters in a "mitigation" state in which an existing slave replica is promoted to master replica for each object which had the master replica on the failed node, and a new slave replica is created for each object that had a replica on the failed node.

All objects having the master replica on the failed node will be temporarily having one less slave replica. The procedure for election of a new slave replica shall be by random selection, from the subset of nodes still accessible. The runtime provides a mechanism ensuring the creation of the copy on the newly recruited slave node. When all channels are restored, nodes shall first re-synchronise their object dictionaries, and then proceed to transmit in order, all the messages queued locally.

There are applications for which it is desirable to maintain a node available to its clients when it has become isolated. There are other applications, in which an isolated node must completely suspend its operation until the communication is restored. The reference DOORS implementation implements the former behaviour, by maintaining isolated nodes in operation and successfully executing any incoming service requests. All outgoing service requests are queued for an unlimited period of time, and sent to peers as soon as communication is restored.

When all the nodes return and the system is restored, synchronisation shall start only after the mitigation operations are completed. We expect therefore some delay between "physical restoration" of system integrity and "logical restoration" of global system state. The degraded mode of operation due to cascading failures (another node or set of nodes become inaccessible while the node is performing mitigation operations), when the node must abandon current operations and immediately start the mitigation specific to the newly detected state is not further researched, as DOORS is not expected to continue running in such conditions.

We measured the cost of replication and its evolution as a function of the replication factor and of the number of objects. We used a geometric progression in the number of objects: 1000, 2000, 4000, all the way up to 512 000 objects. We performed 2 sets of measurements for replication factors of 2 and 3. The results are depicted in Figure 3.6:

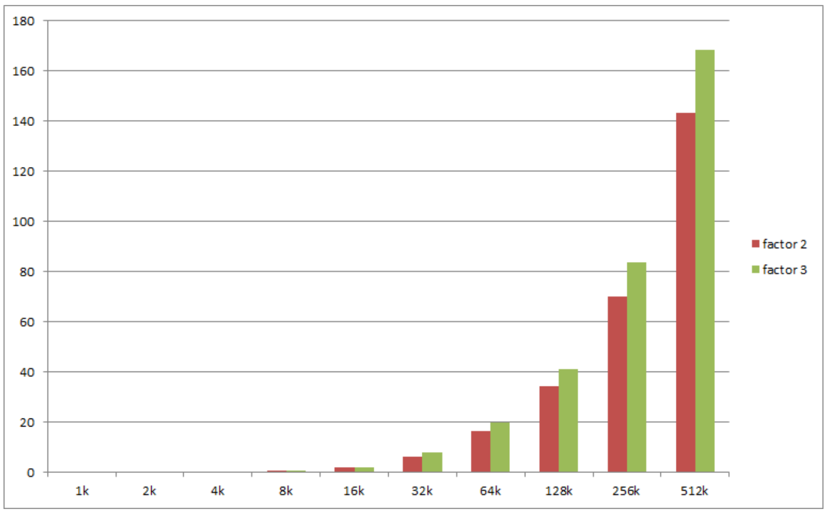

Figure 3.6. Election of new master for unavailable objects.

We noticed a quasi-linear increase in duration, as according to the chosen data architecture, the election of a new master per object should is O(n). The measured cost was less than 170 ms per election for 512 000 objects.

3.5 Partitioning

Partitioning consists in spreading out the objects over the available nodes, such that element of system state belongs to a single partition. This achieves horizontal scaling, and covers both computing capacity and storage space. The efficient distribution of objects over the available nodes ensures uniform use of resources and the avoids "computing hot-spots".

Uniform distribution of data over the node space by using hashes and then assigning equal ranges of the hash-space to each node in the system does not require coordination, but complicates the problem of finding where the data is hosted. In this case, queries must be sent to all partitions, and optimisation relies on globally maintained indexes, which need consensus themselves.

By default in DOORS, objects are allocated on the node on which their construction is requested. The class definitions are present all nodes. The DOORS runtime contains mechanisms for explicit partitioning: the client, or even another object, may specify the node on which an object shall be hosted.

The act of optimally placing objects on available nodes is named "balancing". When defining the decision criteria for relocating objects, we evaluate three major capacities: computing (the amount of CPU available), memory, and communication. Each of them has to be measured continuously at node-level and the decision to move an object, as well as the choice of destination node shall be based on the global set of capacities, as shared by all the nodes.

The global problem of balancing is depending on the local problems of deciding how to schedule method execution on the cores available, deciding which objects to keep in memory and which to evict to permanent storage, and deciding how to prioritise messages exchanged by the objects, across the available network interfaces. The dedicated architectural entities addressing each of these are depicted in Figure 3.7:

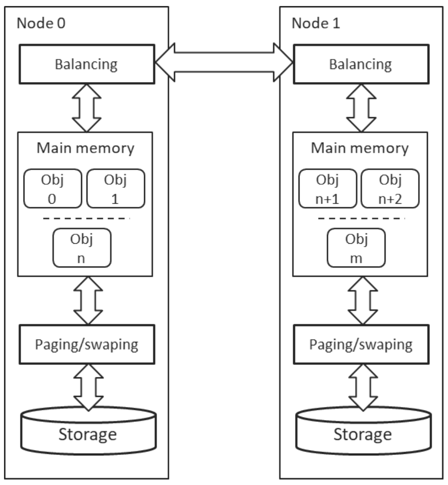

Figure 3.7. Scheduler, balancer and persistence manager in the DOORS node architecture.

There is a high degree of interdependence in this design. Changes in scheduling will impact the persistence management and the amount of needed inter-node communication. Moving objects around nodes will influence the number and size of objects to be scheduled for execution, for communication, and for being stored into memory on each node involved in object migration. Useful insight on the matter of scheduling in heterogeneous environments was extracted from [37] and [32], however, for the reference implementation, we defer the matter of scheduling to the host OS.

In order to emphasise on the relevance of inter-node communication in the context of balancing, we present the summary analysis described by [18]. Even home-use network equipment is able to ensure latencies lower than "traditional storage" (at the moment of this analysis, NVMe storage was not yet mass-market). We draw two partial conclusions:

- if a node is unable to maintain all active objects in its main memory, migration of some of these active objects to another node is preferable to swapping the object out to local storage;
- achieving object locality, in which objects communicate mostly (or even only) with other objects hosted on the same node is the best performance optimisation policy.

An object should be migrated on another node when it is starving for execution time, or there is no available space in the main memory and the object is runnable, or the object is communicating mainly with remote entities (i.e. objects residing on other nodes, for which the latency is relatively large). The balancing algorithm must measure the length of the scheduling queue, the available memory, the latencies and throughputs experienced by the current node when communicating with its peers, and how “far” are the objects targeted by the messages sent by the objects on the current node.

The metrics are structured as series of values. We use of percentiles instead of averages, due to the fact that averages overlook of the outliers. In practice, outliers matter a lot more, and under conditions which are less than exceptional (sometimes even periodically) latency increases or bandwidth drops of more than one order of magnitude. Even if only a small percentage of the messages experience high latency, if the same object issues many messages, its operation ends up executing very slowly, due to the so-called *tail latency amplification* [10]. These outliers will skew the computed averages, as they do not represent typical system behaviour [36]. In contrast, percentile values are robust against the extreme outliers and represent a reliable decision support for object relocation. We found efficient computation algorithms for percentiles: *forward decay* [9], *t-digest* [12], and *HdrHistogram* [35]. The latter appears to be better suited for a real-time application, needed for DOORS.

The DOORS runtime offers introspection. Object structure and content can be inspected at runtime. Elements of the runtime are also exposed as "normal" objects. The services of the system are in fact accessed by sending messages to system objects and the conversion into native method invocations is completely handled by the runtime.

We performed concrete measurements of the cost of object migration a wireless environment, with a nominal speed of 780 Mbps. We used objects of 3 different sizes: 20, 120, and 512 bytes respectively. The sets to be migrated were the already usual sizes, starting from 1000 objects and ending with 128 000 objects. The latencies measured had a mean value of 2743 us and the durations measured (in milliseconds) are depicted in logarithmic scale in Figure 3.8.

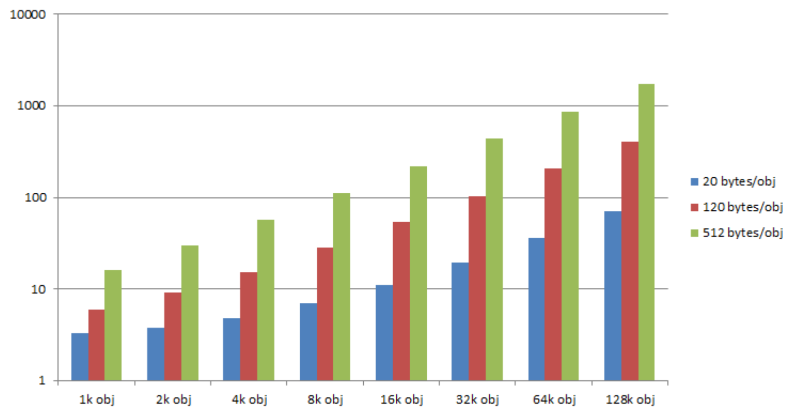

Figure 3.8. Duration of object migration, on logarithmic scale.

The recorded values stem from 3 ms (for 1000 objects of 20 bytes each) to 1.8 seconds (for 128000 objects of 512 bytes each). We expect very few scenarios in which sets larger than a few thousand objects need to be migrated, but we expect to have applications in which the network will have dramatically lower performance and reliability.

3.6 Routing of messages

Routing messages is making sure that they reach their intended destination, in the context of partitioning. DOORS allows the clients to connect to any node and send messages to any object hosted in the system, by having the directly connected node act as a router and implement an "object address book". We use the name *Object Dictionary* for this "address book", as it also contains other important object attributes, relevant to their correct and efficient management. The structure is outlined in Figure 3.9.

When objects are being migrated from one node to the other, the object dictionary is updated. DOORS nodes maintain the convention to promote to master the first element from the list of the secondary replicas (implicit consensus). Other operations which produce modifications of the object dictionary are creation and deletion of objects.

Partitioning may be used in conjunction with replication in DOORS, for increased resilience and capacity.

Figure 3.10 shows an example DOORS instance, containing 4 nodes and using a replication factor of 3. The system hosts 4 objects, and each of them has 1 master replica

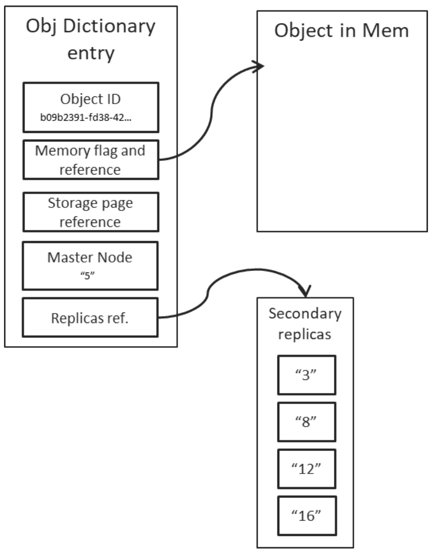

Figure 3.9. Structure of an Object Dictionary entry.

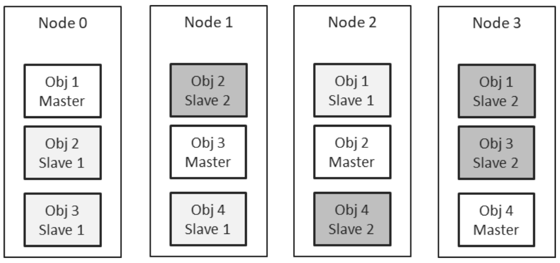

Figure 3.10. Partitioning and replication in a DOORS instance.

and 2 slave replicas.

3.7 Achievable consistency models. Transactions

Transactions are a convenient mechanism to group multiple operations - reads and writes - into a single entity and provide the ACID guarantees for them. The industry-standard relational database products of today do not fully implement serializability, offering instead variants with weaker guarantee, such as *snapshot isolation* [15], or *multi-version concurrency control* [6].

There is a continuum of consistency models offered by real-life implementations of distributed systems. A systematic, view on the subject is given by [4]. The hierarchy shows how stronger consistency models rely on having the constraints of weaker ones as prerequisites.

According to the same paper, the level of availability provided by a system increases as we "descend" through the tree. There are three levels of availability defined: highly available - any non-faulty node, even in complete absence of inter-node communication, "sticky available" - any non-faulty node, as long as the client "stick" to the same node when sending messages, and low available - not available during failures of the communication channels. At least some of the nodes must pause their operation in order to ensure consistency.

The consistency models presented in [4] which provide high availability are:

- monotonic reads - if one process performs reads R1 and R2, then R2 cannot observe

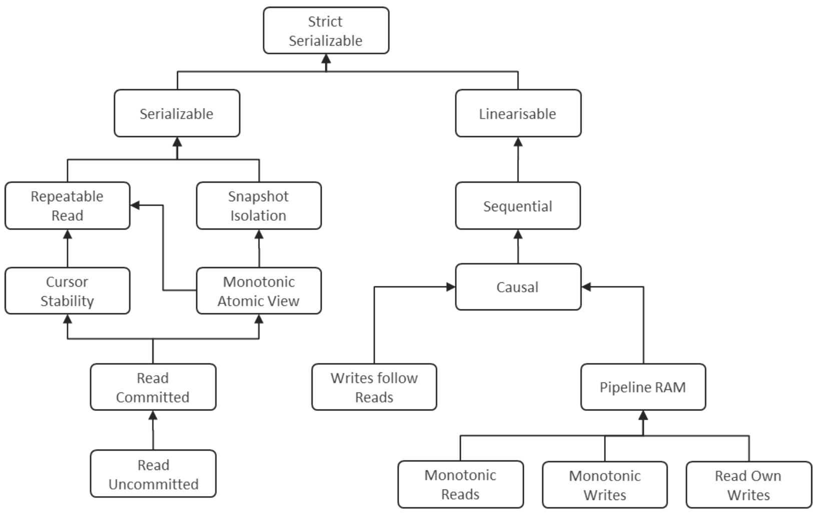

Figure 3.11. The hierarchy of consistency models, as described in [4].

a state prior to the one available in R1;
- monotonic writes - if one process performs writes W1 and W2, then any other process shall observe first W1 and only then W2;
- writes follow reads - if one process reads value V, as written by write W1, and then the same process performs write W2, then W2 shall be visible after W1;
- read uncommitted - everything is possible, except "dirty writes", i.e. if two parallel transactions attempt to modify the same object before committing, one of them shall fail;
- read committed - everything is possible except "dirty writes" and "dirty reads". In addition to the condition above, any write performed by transaction T1 shall not be visible by any other concurrent transaction prior to the commit of T1;
- monotonic atomic view - if transaction T1 performs a write W and transaction T2 observes the effects of W, then T2 shall observe all writes performed by T1.

Here are the consistency models presented in [4] which provide "sticky" availability:
- read own writes - if one process performs write W and later performs read R, then R must observe the effects of W;
- pipeline RAM - the cumulation of "read own writes", "monotonic writes", and "monotonic reads";
- causality - causally-related operations shall appear in the same order on all processes.

The consistency models presented in [4] which provide low availability are:
- sequentiality - the "total order" of operations in the system is consistent with the "partial orders" of the operations on each node;
- snapshot isolation - each transaction T appears to operate on an independent, consistent "snapshot" of the state of the system. This snapshot contains the effects of all transactions which were committed prior to the initiation of T;
- cursor stability - when transaction T reads an object using a "cursor", the said object cannot be modified by any other transaction until the cursor is "released";
- repeatable read - when a transaction T reads an object on which it does not perform any modification, it shall read the same object values throughout its life - even if a parallel transaction committed a different value for the object in the meantime;
- linearisability - all operations on a single object appear to take place atomically, in

an order which is consistent with the real-time ordering of the operations;

- serializability - all operations on a set of multiple objects appear to take place atomically, in an order which is the same for all observers, but not necessarily consistent with the real-time ordering of the operations;
- strict serializability - similar to serializability, but also consistent with the real-time ordering of the operations.

There are market-relevant systems which offer less than ACID: these are the so-called *BASE systems*, which mean "Basically Available, Soft-state, Eventually-consistent". It is safe to consider BASE any system which does not offer ACID guarantees.

In DOORS, transactions are sequences of messages exchanged by objects and which may produce state changes. The distributive nature of a transaction depends on the placing of the objects participating in it. The balancing algorithm becomes an important actor in the area of transaction optimization, as by choosing the same node of residence for individual objects, the number of distributed transactions can be reduced.

There is a set of replicated data structures, for which DOORS must offer distributed transactions: the Node Dictionary, the Class Dictionary, and the Object Dictionary. Updates of the latter are the most frequent case of distributed operation. The three possible changes are: (i) Adding new objects on any node; (ii) Deleting existing objects on any node; (iii) Moving existing objects from one node to another. Such changes must propagate on all peers as soon as possible, but not all need to be transactional. Messages sent to an already deleted object would simply fail. Learning a bit later about a new object existence would harm performance and not consistency.

Changes of object state is only possible using messages, which in turn cause execution of object methods. There is only one copy for each DOORS object, therefore single-object transactions are local only.

The simple algorithm that ensures atomic commitment protocol for a distributed transaction is the Two-phase Commit [21]. The disadvantage that needs to be addressed in the concrete implementation is the inherent blocking nature due to the existence of a ”single point of failure”. Transaction coordinator (which, could always be chosen as the node that originates the first message in the transaction) may fail, effectively stopping progress. DOORS can solve this, by relying on events and time-outs, both for individual messages and for entire transactions.

3.8 Concurrency control

Moderation of concurrent control is needed in a system that allows simultaneous read and write access to a resource from more than one interface. This is needed in DOORS even when the system is composed of a single node; several clients may connect to the node and send messages to the same object simultaneously. In the interest of simplicity, we choose to implement S2PL (Strict Two-Phase Locking) in DOORS, on the basis of [5], and disregard the impact of performance, at least in this reference implementation.

DOORS uses MVCC in the management of classes. The reason behind is the proven ability of MVCC to implement *snapshot isolation*. All reads made during a transaction shall see a consistent "picture" of the class types involved. Existence of a class instance is a "transaction" in a very broad sense. As long as that particular instance exists, DOORS shall maintain the description of its structure unchanged. In accordance with the primary design goals of the system, and mentioned in Section 2.3, DOORS offers the possibility to update object structure and behaviour (hence class specification) during normal operation. This is achievable if classes are managed in a MVCC set-up.

How should the system behave when two or more external actors attempt to update a certain object with different end results? Based on the nature of the problem, the acceptable solution may vary, from immediate, automatic and transparent merging of results to halting all progress, cancelling many or all the required operations, or triggering recovery procedures and resuming normal operation only after explicit confirmation. We consider these questions fit to be answered by the application.

The *CAP Theorem* was first stated in 2000 at the Symposium on Principles of Distributed Computing and formally proven in 2002, by Nancy Lynch and Seth Gilbert. It is a mistake to consider it a trilemma. Further study shows that Consistence, Availability and Partition Tolerance have asymmetric roles [7]. All three are achieved, as long as partitioning of the system does not occur. The system must choose between consistency or availability only in the presence of segmentation.

PACELC builds on the CAP theorem and shows that even in the absence of network partitioning, trade-offs need to be made: between consistency and latency. This is formalized in [1], and the reason for this inevitable trade-off is the fact that a distributed system that ensures availability must always replicate data, and replication needs extra time (latency). PACELC therefore stands for ("Partitioning then Availability or Consistency, Else Latency or Consistency").

3.9 How eventual is eventual consistency?

Evaluating the CAP/PACELC theorem, we conclude that latency in propagating state changes to replicas is inevitable. Whenever replication is in place, there is a time period in which replicas are not in sync. We also state that "eventual consistency" is in fact the only form of achievable consistency. Any form of consistency is "eventual", the difference is made by the time needed for convergence, and this time is relevant only when compared to the liveness needs of the problem solved by our DOORS implementation. Furthermore, the boundary between high latency and connection loss is many times open to interpretation. It is the receiver's decision to consider whether the late data is still useful, or relevant. There are business cases in which the systems to be modelled/supported have hard real-time constraints, and if a certain piece of prerequisite data is not received in a well-defined window of time, any transaction depending on it becomes obsolete and therefore has to be aborted. Any latency larger than the said window of time shall be considered "connection loss". In order to be useful, DOORS shall make sure to achieve consistency and disseminate the newly achieved consistent state within the boundaries of this window (we name it*window of liveness*). The system shall implement measurements and estimations, in order to determine which is the smallest window of liveness achievable in a particular DOORS implementation. Once obtained, these metrics could be used by the application developer in deciding the suitability of a particular implementation, optimisation opportunities, scheduling policies, etc.

4 | Node Level Capabilities

DOORS features presented in the previous section rely on the following capabilities present at node level:

- sustain real-time, asynchronous, bidirectional communication;
- detect the situation in which messages were not received in timely fashion;
- interpret messages as notifications or as service requests;
- execute computation as response to service requests;
- specify object structure and behaviour (class definitions);
- create individual objects based on a class definition (instantiation);
- create relationships among distinct objects (references);
- detect exceptional conditions and generate corresponding immutable objects (events);
- define behaviour as reaction to events and execute it accordingly (event handlers);
- expose object structure and behaviour in a uniform way (introspection);
- track and store object state (persistence);
- moderate concurrent access to objects (concurrency control).

We envision dedicated components for each of the above capabilities and depict them in the DOORS node architecture, in Figure 4.1.

4.1 Node communication

All node interfaces are bidirectional, TCP based, asynchronous, with time-outs. If payload is not received before the time-out is reached, the interface generates a "time-out" event returns in IDLE state, and discards the received partial message. The employed state machine is illustrated in Figure 4.2.

This solution employs event-based processing and uses an existing, portable communication library, *libevent*, which provides asynchronous event notification, comes with its own event loop and runs on all popular operating systems.

4.2 Objects in DOORS

DOORS provides encapsulation, with a basic access scheme: all fields are private and all methods are public.

Figure 4.3, shows the structure of objects and classes. The keys in the map are the names of the attributes, while the values in the map are dedicated structures, as depicted in Figure 4.4. Each attribute contains therefore a description, composed of a name, a type, and a value.

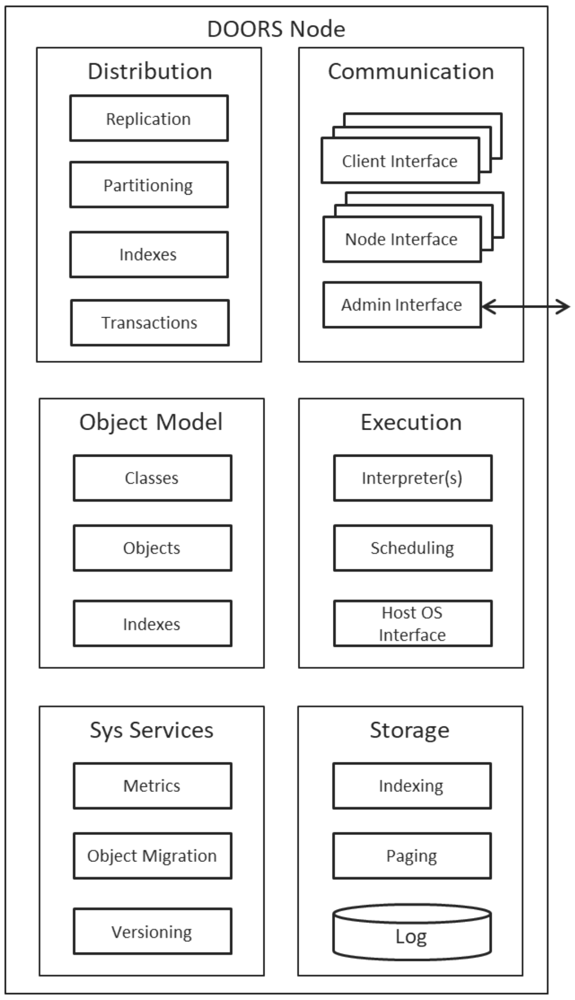

Figure 4.1. Reference architecture for a DOORS node.

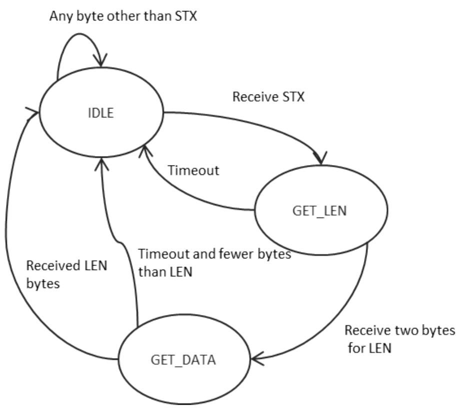

Figure 4.2. Communication state machine.

4.2.1 Defining structure. The DOORS type system

The definition of class structures in DOORS is based on a simple type system, with "fundamental types" and "object references". DOORS defines the following fundamental types: boolean, character, integer, and double. Method parameters, as well as object attributes use composite structures, containing the type designator and the value (stored

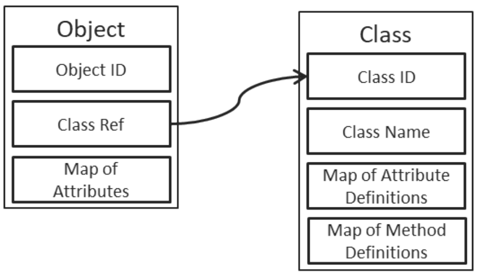

Figure 4.3. Structures of objects and classes in DOORS.

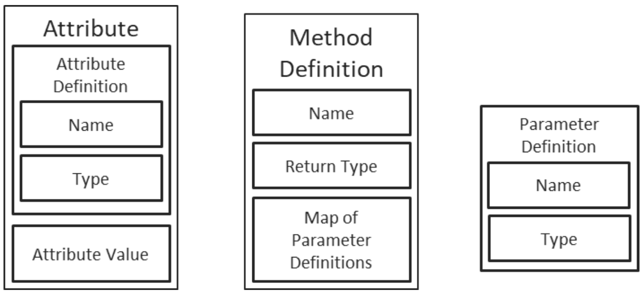

Figure 4.4. Definitions for attributes, methods and parameters.

in an union), as described in Figure 4.4.

DOORS adheres to the Smalltalk [20] philosophy: when two objects exchange one message, the receiving object is the sole responsible of how it responds to it. This process is called either *dispatch* or *binding*. We chose not to include polymorphism in the reference implementation. When parameters are passed in a message, they are being passed "by reference", even when they refer objects of primitive types. Methods that receive the message parameters may therefore have side-effects. Return values are the responses to the service requests, and have two parts: the status - either success or failure - and the actual result. References to objects are their respective GUID, and therefore may pass unchanged from machine to machine.

For efficiency reasons, returning references to objects is preferable, and this is the only way objects exchange other objects between them, within the boundaries of the DOORS instance. However, at the system boundary, a client entity must be able to retrieve the actual object "by value", in serialised form, based on its reference. In order to be able to transfer objects "by-value", the DOORS runtime provides a serialisation service, which "flattens" the maps contained by the serialized entity.

There is no garbage collection in DOORS. Even when there are no variables referring to it, any object will continue existing in DOORS. Visibility of objects is dictated by the scope of the variables referring to them. Local variables will offer local visibility to the referred object.

4.3 DOORS system services

The DOORS runtime operates at node level by providing services, which are primarily made available to external clients via the Client Interface, but they are equally available to the objects hosted within the system. Objects can therefore create other objects and send messages to them. The most important services offered by the DOORS runtime are:

- enumeration of already defined classes;
- retrieval of class specification;

- enumeration of class instances;
- retrieval of the current state of an object;
- serialization and de-serialization of objects;
- sending of messages to objects;
- creation, update and deletion of classes;
- creation, update and deletion of objects.

4.4 Administration Services

Administration of DOORS is performed by properly authorised external entities, which connect to the admin interface of any node in a DOORS instance. The following services are available:

- visualization of node status: size of storage areas, number of objects, clients connected, node ID, number of peers, connection status to each peer, latest throughput and latencies on all open interfaces;
- visualisation of logs;
- node joining/leaving a DOORS system;
- changing/reloading of node configuration;
- node shut-down/restart.

In the spirit of uniform access to all components of the system, administration services are exposed as methods or attributes of *system* objects.

4.5 Introspection

The ability to inspect the contents of objects and the interfaces in a simple, unified way is essential in DOORS. It allows for efficient investigation of potential problems, and prepares the environment for fully reflective behaviour, in which the system may be modified while running. The entire runtime is exposed over the Admin Interface as normal DOORS objects, simplifying the implementation of any DOORS management tool.

The most important features of DOORS which allow for the implementation of introspection are:

- class modelling: the class dictionary available in each runtime instance contains class specifications for all components relevant to the developer. There are attributes defined for each of them, made accessible in a manner similar with the classes created by the "normal" users;
- object wrapping: in addition to modelling the structure and state of these runtime components, they are also implemented as normal DOORS objects and inserted into the object dictionary on each node. The object querying services defined in Section 4.3 may be used upon them;
- transparent access: the *getter* methods for the attributes of the wrapping objects have implementations which provide access to the underlying data fields of the runtime component.

4.6 Object Persistence, tracking and threading

Objects are being saved to "disk" or other form of persistent storage, in a manner transparent to the client. As not all hosted objects fit into memory, DOORS employs swapping and only keeps in memory the most recently used objects. We shall implement the naive version of the LRU algorithm for the page replacement policy and we consider its performance to be of lesser relevance.

Class descriptions are replicated on each node in a DOORS instance, while objects are partitioned among the available nodes. DOORS uses dedicated data structures in order to keep track of both classes and their instances: the Class Dictionary - the unique IDs and names of classes, as well as their definitions, and the Object Dictionary - the unique IDs and locations for all objects in the system.

Each DOORS object contains a thread. The allocation of each executed method to a core, and the scheduling of the object thread to execution is left at the discretion of the host OS.

4.7 Concurrency control at node level

A single node may receive simultaneous requests from two or more connected clients, which target the same object and may also perform conflicting operations. The following concurrency control approaches are being considered:

- locking algorithms and their variants (e.g. 2PL, 3PL and variants);
- time-stamp ordering (more in the sense of vector clocks and less as real-time labels);
- multi-version concurrency control (a current favourite, and already considered for the management of classes);
- lock-free approaches: compare-and-set/ compare-and-swap and their use in lock free deques.

5 | Application Level Challenges

In this chapter we discuss the challenges present in distributed systems at application level. Distinct applications running on the same system may still interact with one-another and they also need to share access to common resources present on the hosting node, such as hardware devices.

5.1 Deployment and versioning of behaviour and structure

The traditional way of addressing application updates is to decommission the current version of a component, run database scripts in order to update the data structure and then start up the new version of the said component. These are usually performed manually, with checks and smoke tests being performed after each step. At best, a deployment orchestration tool may be used, such as Ansible, which has all deployment and migration operations scripted. DOORS aims to incorporate these capabilities inside the objects themselves as much as possible. Update of an object structure shall be a method of the said object.

As defined by [3], deployment is a process of organisation of a set of activities in order to make an up-to-date software application available. The major events in the lifecycle of an application are installation, update and retirement. We must therefore address the following challenges:

- definition of application boundaries - one application is defined as a set of components, which must be hosted by the environment;
- management of changes - a new version of an application entails changes to most, if not all, deployed components. Deployment of a new version may be seen as a large update transaction;
- management of dependencies among components. Updating the interface of one dependency requires the update of all components depending on it, so on;
- coordination between deployment and use of the application - "hot updates" are seldom possible. The system must provide mechanisms for taking components offline, update and then return them into operation, in the context of component interdependencies;
- content delivery - After addressing the coordination issue and providing application components with special running states allowing for updates, the final matter to be solved is that of the delivery of the new specification, in the presence of slow or unreliable communication channels.

Taking inspiration from [14], which presents a self-deployment protocol, we resort to the introspective nature of DOORS. Applications are objects themselves. A DOORS application shall be a class which implements a set of methods, which we collectively refer to as the *App protocol*. An instance of *App* shall be a "singleton" and shall provide implementations for all methods mentioned into the protocol, which are descriptions of behaviour in case of the major lifecycle events described above: packaging, (inter)dependencies, installation, activation/deactivation, update and deinstallation.

It shall be noted that, as a direct consequence of the fact that DOORS applications are objects themselves, the principles and mechanisms for the balancing of objects apply to them as well. However, in the case of applications, additional criteria for optimisation may emerge. There are application-specific constraints which need to be considered when solving the problem of optimal application deployment. A generic, quantitative approach, where the results are modelled as QoS figures, is described in [25].

In DOORS, we choose to implement a feature which we trust is going to simplify deployment, as well as address the issues emerging during updates: allow co-existence of multiple versions of the same class. Implementation of MVCC at this level ensures snapshot isolation and allows (at least in theory) the complete update of an application without taking it completely off-line. When a client performs the update of the definition of a class, the runtime creates a new version of the class, and distributes this across all nodes. The old version of the class description is not deleted. It is however marked as "non-actual" and the new version is appended to the class dictionary, and a reference to the previous version is created. The DOORS runtime shall always use the latest version of a class when creating new objects, and the object instances are not implicitly updated to the new version. This needs to be performed explicitly by the developer, by writing dedicated methods (thus implementing the required *protocol*.

5.2 Modelling of events and their processing

Events are immutable. They state that a certain point in time (and space) something happened. That particular event tells about a change in the state of the universe in that particular point in space and at that particular moment in time. The most straightforward way of modelling an event in a system is a tuple. The set of event coordinates are the individual values stored in the tuple. Successions of events are stored and processed in ordered form. In many applications, particular sequences of events have special significance and must be also processed. We therefore get new kinds of events, as results of a class of operations called Complex Event Processing. An important feature of CEP systems is that they need to support streams of event arrivals. The system must describe how secondary data, decisions and insights are derived from these.

In most of these real-life applications, the input is unbounded. A data acquisition system gathering data from remote industrial processes must be able to acquire, interpret and store data points "forever". There is no concept of "last data point" or "last process event". The abstraction which allows us to formalise such a system is the so-called "Stream Processing". Events become available incrementally, they may be interpreted, in the sense that partial sequences are being automatically analysed and other data items are being derived, and they may either be stored or discarded, based on the concrete needs of the problem solved by the implemented system. An example is depicted in Figure 5.1.

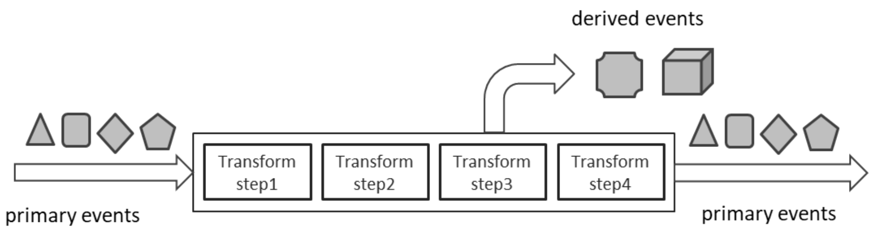

Figure 5.1. Processing of event streams.

The concepts of producers, consumers and streams maps natively to DOORS. This is because they map directly to DOORS design entities. Objects communicate with each-other exchanging messages. One object is the producer, the outgoing messages (as immutable, relatively small pieces of data) may be used directly to model "events", while the recipient of the message would be the "consumer", receiving and processing them asynchronously.

Industry standard message queueing systems allow for definition of message management policies and their central component is always some form of message broker. Multiple objects may subscribe to the same topic and consume the same series of messages. Popular specifications for broker behaviour are available in JMS [19] and AMQP [2], and they allow for multiple consumers and they exhibit configurable behaviour, allowing for either load balancing - when each message is delivered to a single consumer, out of potentially many, sharing work, or "fan-out", when each message is delivered to all consumers - effectively implementing the "event broadcast" function.

The inherent "one object-one message queue" limitation of DOORS may be avoided, and broker behaviour may be implemented in DOORS by encapsulating it in a dedicated object. Such an object shall act as intermediary between the producers and the consumers, offer internal data structures for storing incoming messages prior to relaying them to the final consumers, expose an interface allowing for topic creation, publishing into topics an subscribing to topics, and implement message processing policies for load balancing or fanning out messages.

At the forefront of Complex Event Processing sits the relatively new domain of "Stream Analytics", which is a statistical interpretation of CEP. Instead of concentrating on searching for event patterns, Stream Analytics focuses on aggregating events into statistical metrics and series of derived data. In addition, while CEP is usually operating with relatively short sequences of events (few individual, and up to tens of events), Stream Analytics is geared towards operating with long sequences (sets of hundreds, thousands, or even millions of events). In the realm of Stream Analytics we encounter probabilistic algorithms, such as HyperLogLog [17] for cardinality estimation, percentile calculations (as described in Section 3.5, and Bloom filters applied for determining set membership.

Originating from the world of databases, Change Data Capture is a technique in which the changes performed to the data are being recorded into log-type structures, which are then made available to clients, possibly as event stream. If we have the original state of the system, we can construct the current state of the system by "replaying" all the changes which occurred over time, effectively replicating the final state of the data in another data storage. In the world of Domain Driven Design, the CDC has been evolved into the concept of event sourcing. The primary improvement is orientation towards business-meaningful transformation. Instead of following elementary modifications to data articles, event sourcing operates with immutable units (business-specific events) which are interpreted, transmitted, stored and "played" in order. The event logs are "append-only", nothing gets deleted, and the system state at a certain moment may be computed by interpreting the individual events in the order of their arrival.

5.3 Application-level security

DOORS clients shall be authenticated, at connection level, ensuring the authenticity and non-repudiation of all messages received via the respective connection. For the authorisation, the system must keep a record of all users and their level of authority. This is completely application specific, so the system shall only provide means of enforcing authorisation. Objects in DOORS inherit the authority level of their creator. If dictated by the application needs, the objects may explicitly relinquish some of their privileges, opting for fewer rights. In contrast with external clients, the objects themselves need not be authenticated. The creation of objects is itself a "privileged operation" and we consider it sufficient to ensure system integrity, at least in order to fulfil the academic goals of the DOORS implementation.

We expect that DOORS shall be operated in trusted physical environments. All nodes joining the network in which DOORS operates are considered "authentic" and by default allowed to join the DOORS instance. The system may be further secured by encryption at transport level and use key-pair-based technologies. All nodes are "full-right peers", therefore no authorisation mechanism for entire nodes shall be implemented in DOORS.

Rights in DOORS are inspired from the *rwx* bits, as defined in POSIX environments. In a similar manner, each object shall be endowed with right attributes for: receiving and processing messages (analogue to "execute"), exposing its specification (analogue to "read") and allowing for modification of specification (analogue to "write").

6 | Case studies and practical considerations

6.1 Case study - Telemetry

There are standards regarding the quality of the power circulated through various points of the grid, such as EN50160, described in [27] and IEC 61000-4-30, exemplified in [23], describing which measurements need to be done, how often, and how they need to be interpreted and aggregated in order to quantify the quality of the delivered energy. There are specialised devices which are capable of monitoring the power lines (typically in substations, but they may be installed at consumers sites as well), acquiring and aggregating the needed parameters and then passing them to central nodes, where this data is being further aggregated into synthetic and analytic reports, used further in the maintenance of the grid, and delivered to authorities as part of the normal monitoring processes at national level.

The power quality parameters are either r.m.s. values, computed over regular intervals or power quality events, such as voltage interruptions, dips and swells, harmonic distortion events, etc. Another type of quality data are the fault recordings. These are "snapshots" of the voltage and/or current recorded in various points of the grid, when certain power parameters cross predefined thresholds.

In a typical distribution network, there are tens of substations, distributed across hundreds of kilometres. There are "primary aggregations", performed by the dedicated acquisition devices or by the nodes hosted in the regional centres, and "secondary aggregations", typically performed at the hub of the system, by powerful nodes, which may employ redundancy and data replication.

Such a system needs to be maintained up to date and in top shape, via periodic software updates, as well as mandated parameter configuration, to be done at each monitored point.

The design of DOORS fulfils all the requirements of such a telemetry system, and we chose to implement a power quality monitoring system prototype, in order to assess the impact of the balancing framework upon the overall performance of the system.

We assessed the bandwidth and storage needs for a system of real-life magnitude, based on the dimensions of one specific power distributor present in the Romanian electrical grid, which monitors around 580 3-phase systems across its network of over 200 distribution substations. The modelled distribution grid spans over 6 counties and is structure into 3 regional centres (two counties per centre, yielding roughly 200 monitored points per centre).

In addition to the flow of acquired data-sets, starting in each acquisition node and terminating in the central aggregator, we took into consideration the object updates flow, which is initiated by the central aggregator and propagates to all the other nodes in the system. The objects implementing acquisition of data, parameterisation and operation of

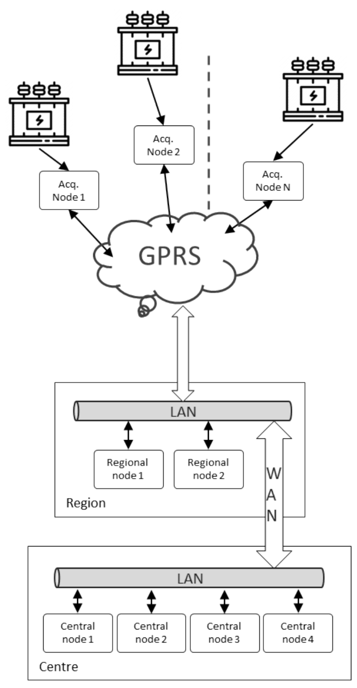

Figure 6.1. The modelled system.

the in-field device do not move from the acquisition nodes. The objects performing event processing, designated in the table as "primary aggregation" and "secondary aggregation", may be relocated.

One iteration in the event processing loop for the primary aggregation lasts less than 100 ms, and this needs to be ran less than 500 times per day (the usual number of PQ events received daily). The cost of one migration of the primary aggregator objects from the acquisition node to the regional node is almost one order of magnitude larger than the entire communication cost of the primary event processing function. These findings need to be considered when deciding on which node type to place the primary aggregator objects. Other important factors to consider are:

- the reliability of the communication channel: in most real-life implementations, the stability and consistency of the connection between the acquisition nodes and the regional nodes is rather low, due to geographical and weather conditions influencing the GPRS link;
- the power constraints that need to be fulfilled by the acquisition nodes. Critical equipment running in substations need to maintain autonomy in the event of total power loss, and have a lower priority and relevance than safety-related equipment, or other operation-critical devices, which are also permanently connected to the site batteries. The acquisition nodes must be as frugal as possible;
- the bandwidth remaining available, after allocating enough for the "productive" data exchange. In our case, one acquisition node generates up to 9MB of data per day, which requires less than 1 hour to be completely transferred into the corresponding regional node.

The system under analysis allows for several migrations of the primary aggregators per day but, for efficiency reasons, the number of object migrations could be limited to one, up to a few per day. A custom-tailored solution, built by the system implementer based on the concrete constraints present in the deployment environment, is simpler, more reliable and easier to test/validate than a general-purpose, decision agent built using meta-heuristic or probabilistic techniques.

6.2 Case study - Edge Computing

Edge computing is concerned with bringing processing resources closer to the requests. This improves latency and shortens the communication paths, therefore increasing the resilience of the entire system. The earliest form of edge computing is represented by content delivery networks, where static resources are pushed closer to the clients in order to avoid capacity issues at the centre.

The concept of edge computing remains valid also in the cases where data needs to flow in both directions in more equal proportions, or when the content to be shared is less static. In fact it becomes even more relevant, as new challenges emerge in maintaining consistency and arbitrating modifications. Examples are:

- autonomous vehicles - where compute-intensive tasks such as image processing and obstacle recognition are being off-loaded by the vehicles to more powerful, sedentary nodes;
- smart grids - where sensors placed on machinery are used to measure consumption and generated disturbances and dedicated controllers are able to schedule the running times of high-powered equipment during off-peak periods;
- patient monitoring - in hospitals, where sensors are used to monitor health parameters, and use the local hospital network to notify medical staff in cases of emergency;
- traffic management - even when not considering autonomous vehicles, the traffic management systems are distributed systems in which edge computing applies, especially with regard to the zoning of decisions, aggregating the inputs from traffic detectors and elaborating traffic light schedules according to local conditions;
- smart homes - the myriad of IoT devices running and acquiring data in smart home deployments benefit from edge computing in implementing local control loops and also distributing the computing tasks related to event aggregation and data logging.

This case study covers maintenance in railway systems. These wide-area distributed systems need to fulfil extremely stringent requirements with regard to safety, availability and capacity and they have a well defined architecture, evolved over more than 100 years.

The central sub-system is the "Inter-locking", which is charged with making sure that movements of railway vehicles along the tracks, junctions, and crossings are always safe. Interlocking systems rely on highly redundant hardware and network architectures, and a comprehensive set of inputs and outputs. Train presence on various track segments is detected via specialised sensors, and outputs consist in change commands sent to switches and turnouts, line-side signals, and level crossing barriers.

DOORS has not been designed to operate within the real-time, redundancy and capacity constraints required by the core operational systems; however, its distributive nature, along with the ability to reach consensus, encapsulate state and behaviour and migrate it between the available nodes makes it suitable to implement an edge computing based solution for predictive maintenance. An example of monitoring scenario is for the wear of track switches. The electro-mechanical equipment contains sensors and measures a comprehensive set of parameters during each manoeuvre: voltages, currents, durations, movement distance, temperatures, etc.and records time-series for each. By analysing these time-series, a predictive maintenance system may decide the optimum time for performing replacement or maintenance operations, and even generate work instructions for the field personnel performing the maintenance.

As there are hundreds or even thousands of event-generating devices to monitor, we conclude that such a system must be able to ingest and process series of millions of individual events. This may be addressed by creating station-level aggregation nodes, which shall perform the primary processing of the acquired data streams and generation of derived events, in a manner similar to the first case-study - the power quality monitoring. In addition, the data sets must follow the lifecycle of the actual assets installed. If a certain switch machine is being serviced or replaced, the corresponding time series must be reset accordingly, via dedicated commands, travelling from the centre to the periphery.

7 | Conclusion

7.1 Summary of contributions

DOORS achieves the goal of being an architectural simplification of the currently available solutions providing support for edge computing.

Based on the message-passing paradigm, DOORS offers a self contained solution for a large set of distributed problems, and addresses, in the same package a set of challenges which are usually deferred for later in the operation: the extensibility of the system and ability to update while in operation. There is a small set of fundamental features: a simple and reliable solution of exchange of asynchronous messages, the ability to abstract real-life events and treat them as "first-class citizens" in the system, manage both state and behaviour in a uniform way, being able to replicate entities on multiple nodes as well as to partition them over the entire set of nodes available, and offer a clearly defined consistency model. In order to guarantee "at least once" delivery, each message comes with its own confirmation, and the design has a "synchronous" side, using timeouts for liveness.

The traits defined node level are the interface specifications, scheduling mechanisms and policies, object model and representations, etc. The relevant features at system level are described in the largest chapter of our work. This includes replication and partitioning and, in the context of both, we analysed the potential for providing distributed transactions.

At application level we study the ability to version structure and behaviour, the ability to evolve during normal operation, opportunity for abstracting away events and streams of events and, last but not least, ensure secure access and operation of the application.

The potential impact of DOORS in concrete applications sums up to:

- significant simplification of application installation and update;
- minimisation of "in-field" activities for application deployment - it is expected that the need to physically access remote sites in order to perform updates of the software shall be minimal;
- architectural simplification - there is no longer need for separate database engines, used in conjunction with messaging middleware and execution containers;
- cost efficiency - obvious savings achieved by a small technological footprint, the object balancing capabilities, and a more rational use of the available hardware resources.

7.2 List of publications

The results of our research have been presented in the following publications:

1. Dorin Palanciuc. DOORS: Distributed object oriented runtime system (position paper). In 2017 16th International Symposium on Parallel and Distributed Computing (ISPDC) - (Conference paper, ISI);

2. Dorin Palanciuc and Florin Pop. A distributed, object oriented run-time and storage system, framework proposal for edge computing. In BDA 2018 34-ème Conférence sur la Gestion de Données – Principes, Technologies et Applications, Bucarest, 22-26 octobre 2018 - (Poster, Short conference paper, indexed by INRIA Digital Library - https://hal.inria.fr/BDA2018);

3. Dorin Mihai Palanciuc Mawas. Analysis and Design of DOORS, in the Context of Consistency, Availability, Partitioning and Latency. In 2018 21st IEEE International Conference on Computational Science and Engineering - (Conference paper, ISI).

4. Dorin Palanciuc and Florin Pop. Balancing objects on DOORS nodes. In 2021 23rd International Conference on Control Systems and Computer Science (CSCS) - (Conference paper, ISI);

5. Dorin Palanciuc. Implementing replication of objects in DOORS - the object-oriented runtime for edge computing. *Accepted for publication in the Special Issue "Cyber-Physical Systems - from Perception to Action" of the MDPI Sensors Journal* - ISSN 1424-8220 - (Journal article, ISI, Q1).

7.3 Future work

We consider migration of client connections as an extension of object balancing. If client connected to node A ends up sending messages mostly to objects hosted on node B, it is preferable to have it connected directly to node B and therefore relieving node A of the burden of forwarding messages.

Implementation of lock-free data structures and their incorporation in the Object and Class Dictionaries is another important performance optimisation. Using trees instead of linked lists for storing the key sets, or implementing fast sorting algorithms for index (re)creation are examples in this respect.

Being able to hide diverse implementations under simple and uniform interfaces is highly desirable in any system. Polymorphism shall be incorporated into the DOORS design.

The DOORS design relies on the simple but yet to be proven how useful constraint of one object having a single execution thread. In order to have this scale, fibres would be a lighter and more scalable alternative, and there are currently enough successful implementations to ensure our success.

Being a thoroughly asynchronous environment, DOORS is difficult to master in a "traditional" language. We contemplate being able to concisely and intuitively write multi-threaded code, and such is the case of *futures* and *promises*.

The trade-off configurations between consistency and availability are situated on a continuum, and is very difficult for a system designer to know in advance which exact trade-off configuration will make his system proposal most relevant, and useful. We investigate the possibility to provide a form of "consistency a la carte", in which the system implementer may choose how much off-line availability is provided.

Few of the industry standard distributed solutions have been formally validated, and yet we trust them, based on the success recorded on the market, maturity and the abundance of documentation. We shall incorporate into the design all the "test harness" elements, such that testing and validation of various components can be done "as in production". Examples: testing of the consensus mechanism, the resistance to deadlock of the concurrency control mechanism, the behaviour and performance of the transaction arbiter, etc.

Bibliography

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, 2012. doi: 10.1109/mc.2012.33. URL https://doi.org/10.1109/mc.2012.33.

[2] AMQP. The advanced message queuing protocol, 2014. URL https://www.amqp.org/resources/specifications.

[3] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software*, 103:198–218, 2015. doi: 10.1016/j.jss.2015.01.040. URL https://doi.org/10.1016/j.jss.2015.01.040.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions. *Proceedings of the VLDB Endowment*, 7 (3):181–192, 2013. doi: 10.14778/2732232.2732237. URL https://doi.org/10.14778/2732232.2732237.

[5] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981. doi: 10.1145/356842.356846. URL https://doi.org/10.1145/356842.356846.

[6] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983. doi: 10.1145/319996.319998. URL https://doi.org/10.1145/319996.319998.

[7] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*. ACM Press, 2000. doi: 10.1145/343477.343502. URL https://doi.org/10.1145/343477.343502.

[8] Radu-Ioan Ciobanu, Catalin Negru, Florin Pop, Ciprian Dobre, Constandinos X. Mavromoustakis, and George Mastorakis. Drop computing: Ad-hoc dynamic collaborative computing. 92:889–899, 2019. doi: 10.1016/j.future.2017.11.044. URL https://doi.org/10.1016/j.future.2017.11.044.

[9] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. Forward decay: A practical time decay model for streaming systems. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, March 2009. doi: 10.1109/icde.2009.65. URL https://doi.org/10.1109/icde.2009.65.

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. doi: 10.1145/2408776.2408794. URL https://doi.org/10.1145/2408776.2408794.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo. *ACM SIGOPS Operating Systems Review*, 41(6):205–

220, 2007. doi: 10.1145/1323293.1294281. URL https://doi.org/10.1145/1323293.1294281.

[12] Ted Dunning and Omar Ertl. Computing extremely accurate quantiles using t-digests, 2014. URL https://github.com/tdunning/t-digest.

[13] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989. doi: 10.1145/66926.66963. URL https://doi.org/10.1145/66926.66963.

[14] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, and Noel De Palma. Reliable self-deployment of distributed cloud applications. *Software: Practice and Experience*, 47(1):3–20, 2016. doi: 10.1002/spe.2400. URL https://doi.org/10.1002/spe.2400.

[15] Alan Fekete, Dimitrios Liarokapis, Elizabeth ONeil, Patrick ONeil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005. doi: 10.1145/1071610.1071615. URL https://doi.org/10.1145/1071610.1071615.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi: 10.1145/3149.214121. URL https://doi.org/10.1145/3149.214121.

[17] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[18] Brendan Gregg. *Systems Performance. Enterprise and the Cloud.* Prentice Hall, 2013. ISBN 978-0133390094.

[19] JMS. The java messaging service, 1998. URL https://www.oracle.com/java/technologies/java-message-service.html.

[20] Alan C. Kay. The early history of smalltalk, 1996. URL https://doi.org/10.1145/234286.1057828.

[21] Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '95.* ACM Press, 1995. doi: 10.1145/212433.212468. URL https://doi.org/10.1145/212433.212468.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi: 10.1145/359545.359563. URL https://doi.org/10.1145/359545.359563.

[23] Andres E. Legarreta, Javier H. Figueroa, and Julio A. Bortolin. An IEC 61000-4-30 class a power quality monitor: Development and performance analysis. In *11th International Conference on Electrical Power Quality and Utilisation.* IEEE, 2011. doi: 10.1109/epqu.2011.6128813. URL https://doi.org/10.1109/epqu.2011.6128813.

[24] Mihai Letia, N. Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *Operating Systems Review*, 44(2):29–34, 04 2010. URL=http://dx.doi.org/10.1145/1773912.1773921.

[25] Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012. doi: 10.1109/tse.2011.3. URL https://doi.org/10.1109/tse.2011.3.

[26] Radu-Corneliu Marin, Alexandru Gherghina-Pestrea, Alexandru Florin Robert Timisica, Radu-Ioan Ciobanu, and Ciprian Dobre. Device to device collaboration for mobile clouds in drop computing. IEEE, 2019. doi: 10.1109/percomw.2019.8730788. URL https://doi.org/10.1109/percomw.2019.8730788.

[27] Carlo Masetti. Revision of european standard EN 50160 on power quality: Reasons and solutions. In *Proceedings of 14th International Conference on Harmonics and Quality of Power - ICHQP 2010*. IEEE, 2010. doi: 10.1109/ichqp.2010.5625472. URL https://doi.org/10.1109/ichqp.2010.5625472.

[28] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. CRATE. In *Proceedings of the 25th International Conference Companion on World Wide Web - WWW '16 Companion*. ACM Press, 2016. doi: 10.1145/2872518.2890539. URL https://doi.org/10.1145/2872518.2890539.

[29] Silvia-Elena Nistor, George-Mircea Grosu, Raluca-Maria Hampau, Radu-Ioan Ciobanu, Florin Pop, Ciprian-Mihai Dobre, and Pawel Szynkiewicz. Real-time scheduling in drop computing. IEEE, 2021. doi: 10.1109/ccgrid51090.2021.00087. URL https://doi.org/10.1109/ccgrid51090.2021.00087.

[30] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work - CSCW '06*. ACM Press, 2006. doi: 10.1145/1180875.1180916. URL https://doi.org/10.1145/1180875.1180916.

[31] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011. doi: 10.1016/j.jpdc.2010.12.006. URL https://doi.org/10.1016/j.jpdc.2010.12.006.

[32] Andrei Sfrent and Florin Pop. Asymptotic scheduling for many task computing in big data platforms. 319:71–91, 2015. doi: 10.1016/j.ins.2015.03.053. URL https://doi.org/10.1016/j.ins.2015.03.053.

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[34] Marc Shapiro, N. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 01 2011. URL=http://hal.inria.fr/inria-00555588.

[35] Gil Tene. Hdrhistogram, 2013. URL http://hdrhistogram.org.

[36] Tyler Treat. Everything you know about latency is wrong, 2015. URL https://bravenewgeek.com/everything-you-know-about-latency-is-wrong/.

[37] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. 51:61–71, 2015. doi: 10.1016/j.future.2014.11.019. URL https://doi.org/10.1016/j.future.2014.11.019.

[38] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010. doi: 10.1109/tpds.2009.173. URL https://doi.org/10.1109/tpds.2009.173.