2
Background

This chapter presents the main concepts to provide a better understanding about the work described by this thesis. Since the prototype implementation of the approach proposed by this thesis was developed using concepts and techniques of Autonomic Computing and the MAPE-K loop, both are explained in the section 2.4.

2.1
DDS

The Data Distribution Service for Real-Time Systems (DDS) is a standard managed by OMG [14] (Object Management Group) for Publish-Subscribe communication that aims to provide an efficient and low-latency data distribution middleware for distributed applications [15] [16]. The DDS standard promotes a fully decentralized P2P (Peer-to-Peer) and scalable middleware architecture based on the Data-Centric Publish-Subscribe (DCPS) model. It also supports a large array of Quality of Service (QoS) policies for communication (e.g. best effort, reliable, ownership, several levels of data persistency, data flow prioritization and several other message delivery options) [13] [17].

Publishers and Subscribers of a DDS Domain (the collection of nodes pertaining to a single application), which are named Participants, are containers for *Data Writers* and *Data Readers*, respectively, which exchange typed data through a common *Topic* [16]. Pardo-Castellote, Farabaugh and Warren [18] explain that Data Writers and Data Readers are the primary point for a Participant to publish data into a DDS Domain or to access data that has been received by a Subscriber. Figure 1 illustrates the Publication and Subscription Models as well as how the Topic, Data Reader (DR), Data Writer (DW), Publisher, Subscriber and DDS Domain interacts.

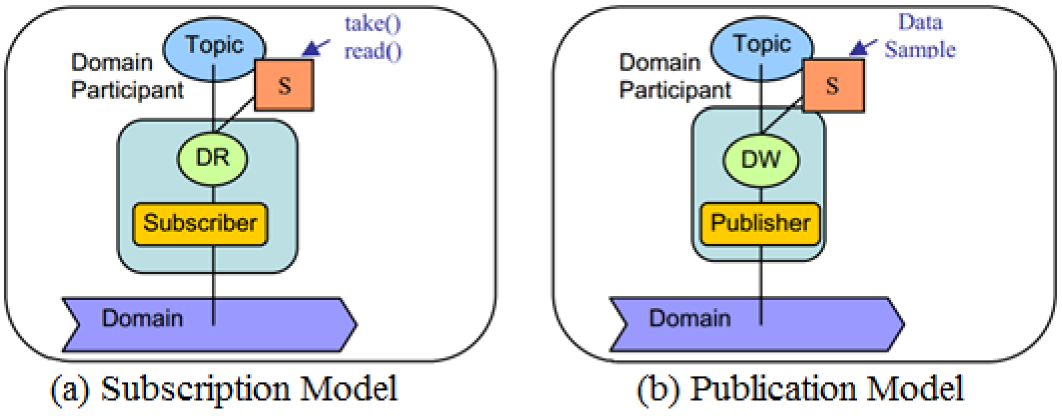

Figure 1 – Publication and Subscription Models [18]

The DCPS makes it possible to organize its Topics in a relational model, providing support for identity and relations, i.e. for each Topic it is possible to define one or more primary keys, and any number of foreign keys representing, respectively, relationships with other Topics.

Unlike traditional Publish-Subscribe middleware, DDS can explicitly control the latency and efficient use of network resources through fine-tuning of its Network Services, that are critical for implementing real-time and soft real-time systems that use QoS policies such as Deadline, Latency Budget, Transport Priority, etc.

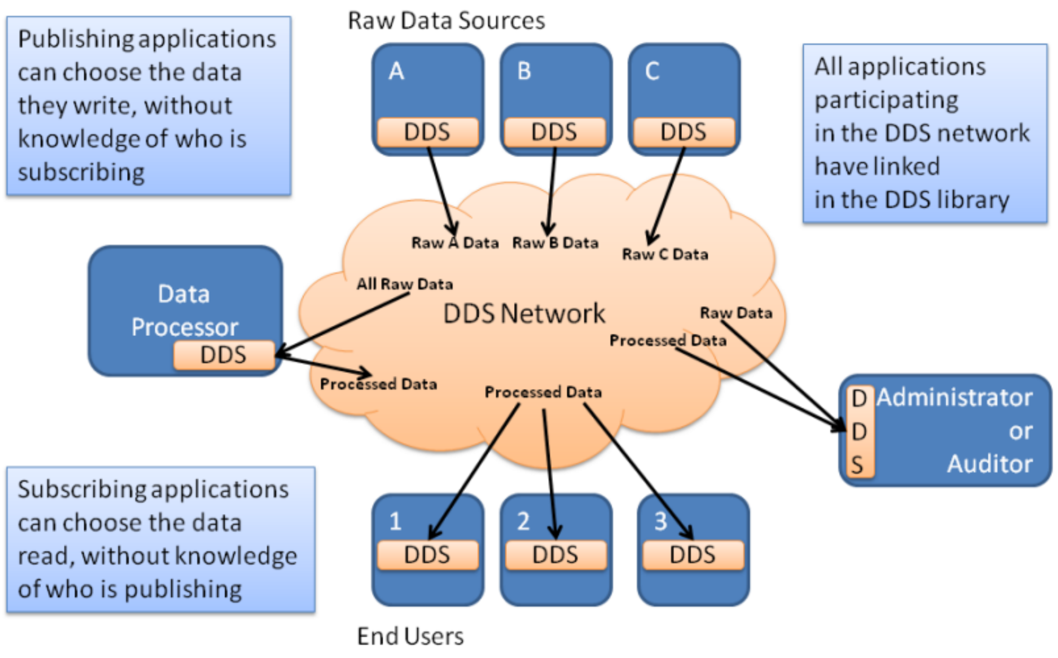
Publishing applications can choose the data they write, without knowledge of who is subscribing
Raw Data Sources
A
DDS
B
DDS
C
DDS
All applications participating in the DDS network have linked in the DDS library
Raw A Data
Raw B Data
Raw C Data
All Raw Data
DDS Network
Raw Data
Processed Data
Data Processor
DDS
Processed Data
Processed Data
DDS
Administrator or Auditor
Subscribing applications can choose the data read, without knowledge of who is publishing
1
DDS
2
DDS
3
DDS
End Users

Figure 2 – DDS System Architecture [16]

Figure 2 illustrates a hypothetical system that uses DDS as a data distribution middleware. This hypothetical application has some sources of "Raw Data", a Data Processor that performs some processing on the "Raw Data" to produce

“Processed Data”, some End Users that consume the processed data, and an Administrative User performing auditing functions, for instance.

DDS supports not only *Topic* subscriptions, but also content-based subscriptions. The latter are enabled by DDS *Content Filtered Topics* which holds a *Filter Expression*, formed through SQL92 (Structured Query Language). This *Filter Expression* defines a selective information subscription, i.e. only the Topic data that match the *Filter Expression* are delivered to the Data Reader. An use example of *Content Filtered Topic* is shown in Figure 3, where a Filter Expression (*Value > 260*) is applied upon the “Value” field. The filters can be applied at the Publisher (*Data Writer*) or at the Subscriber (*Data Reader*). By applying filters at the Publisher, some applications may conserve signification network bandwidth by avoiding the network transmission of irrelevant data [19] to not interested Subscribers. To do so, DDS uses *unicast* to send data instead of using *broadcast* or *multicast*.

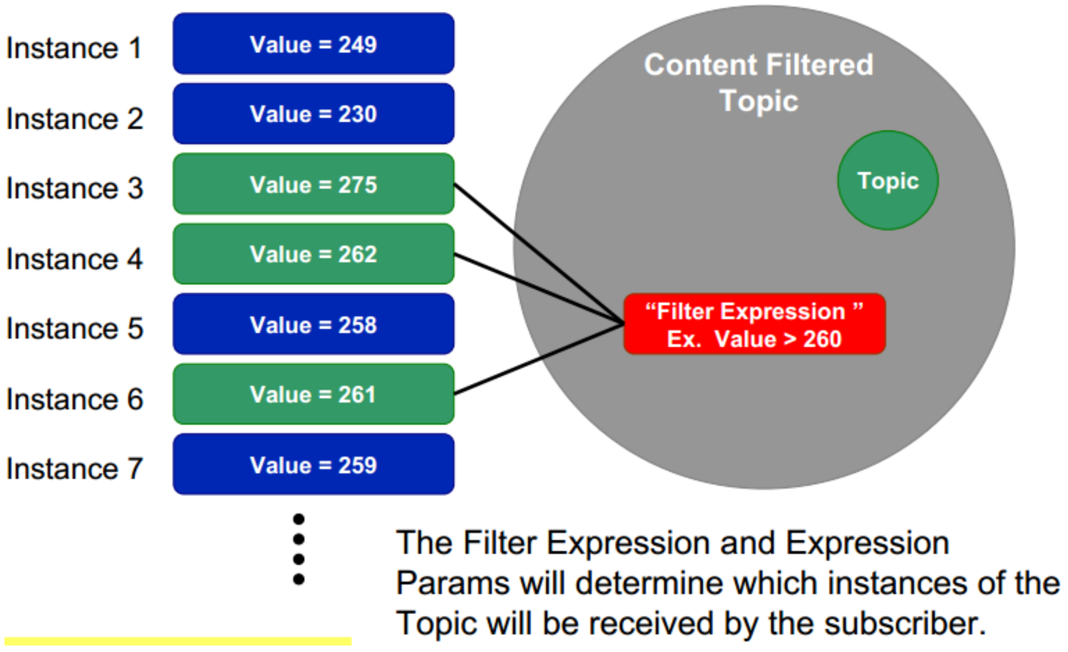

Figure 3 – Content Filtered Topic example [20]

The DDS enables applications to filter data based on the content of the data either at the Publisher side (*Data Writer*) or Subscriber side (*Data Reader*). By applying filters at the Publisher, some applications can conserve signification network bandwidth by avoiding the network transmission of irrelevant data [21]. Although this capability, for some kinds of application – such as those that have a dynamic and unpredictable number of Publishers and Subscribers – the filtering at the Subscribers is the best choice.

2.2
SDDL

SDDL (Scalable Data Distribution Layer) [17] [22] [23] [24] is a communication middleware that connects stationary DDS nodes in wired "core" network to mobile nodes with an IP-based wireless data connection. SDDL employs two communication protocols: Real-Time Publish-Subscribe RTPS Wire Protocol [25] for the wired communication within the SDDL core network, and the Mobile Reliable UDP protocol [23] [17] (MR-UDP) for the inbound and outbound communication between the core network and the mobile nodes.

The core elements rely on DDS' DCPS Model, where DDS *Topics* are defined to be used for communication and coordination between these core nodes. As part of the core network, there are some SDDL nodes with distinguished roles, three of them are: (i) Gateway, (ii) Controller and (iii) *GroupDefiner*.

The Gateway (GW) defines a unique *Point of Attachment* (PoA) for connections with the mobile nodes. The Gateway is thus responsible for managing a separate MR-UDP connection with each of these nodes, forwarding any application-specific message or context information into the core network, and in the opposite direction, converting DDS messages to MR-UDP messages and delivering them reliably to the corresponding Mobile Node(s).

Controller is a control node capable of displaying all the mobile node's current position (or any other context information), and may be applied to manage groups, as well as to send unicast, broadcast, or groupcast message to the mobile nodes.

A processing node is any type of node that performs some processing operation upon the data exchanged. As an example, SDDL has a processing node called *GroupDefiner* that is in charge of evaluating group-memberships of all mobile nodes. To do so, *GroupDefiner* subscribes to the DDS *Topic* where any message or context update from the Mobile Nodes are disseminated and maps each Mobile Node to one or more groups, according to some application-specific group membership processing logic. This group membership information is then shared with all Gateways in the SDDL core network, using a specific DDS *Topic* for this control.

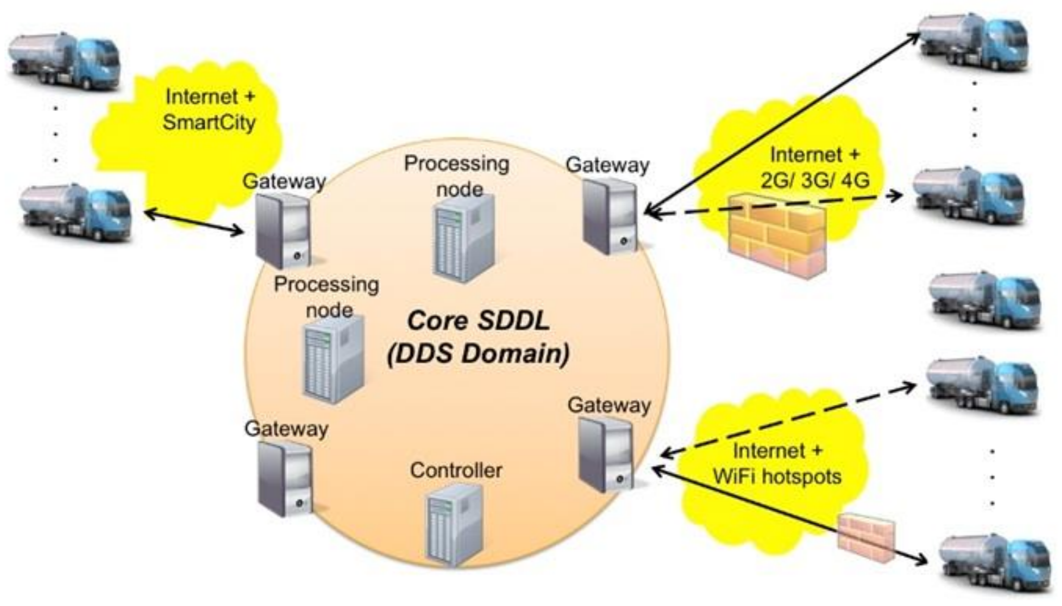

Figure 4 – SDDL Architecture [26]

Figure 4 shows SDDL deployed for a Fleet Tracking and Management application where Mobile Nodes are seen as trucks. However, SDDL is application independent, which means that SDDL can be deployed for any application data.

2.3
Load Balancing in Middleware

With the widespread and rapid development of Cloud Computing and mobile devices, computational resources have become ubiquitous. Despite the recent technology developments, mobile devices still have more restrictive processing and memory capacity and stringent energy limitations than stationary machines. On the other hand, for several applications one has the option to move some processing tasks from the mobile client side to the server/cluster/cloud side. This shift has several advantages for the application, but also increases the demand for load balancing mechanisms [27] [9] [10] [11], especially at the middleware layer used for communication among the servers and/or cluster nodes.

2.3.1
Classification of Load Balancing Algorithms

Load balancing strategies have been classified under a loosely unified set of terms and according to [27], the first classifications came from [28] [29]. Figure 5 depicts the classification proposed by [29]. Following the proposed taxonomy, a load balancing algorithms can be either *local* or *global*. *Local* solutions deal with a single processing node, while *global* algorithms deal with more than one processing node [27]. A global solution may be divided into *static*, when the load balancing algorithm is executed only when there is a new task, and *dynamic*, which runs the algorithm continuously or periodically. At an operational level, an algorithm may be classified as *physically distributed* (distributed) or *physically non-distributed* (centralized). Unlike the centralized approach, in *physically distributed* load balancing algorithms, the decisions are taken by several nodes. And this decision can be made cooperatively, or non-cooperatively. In the former, the algorithm requires a common agreement among the nodes, while in the latter each node makes a selfish decision.

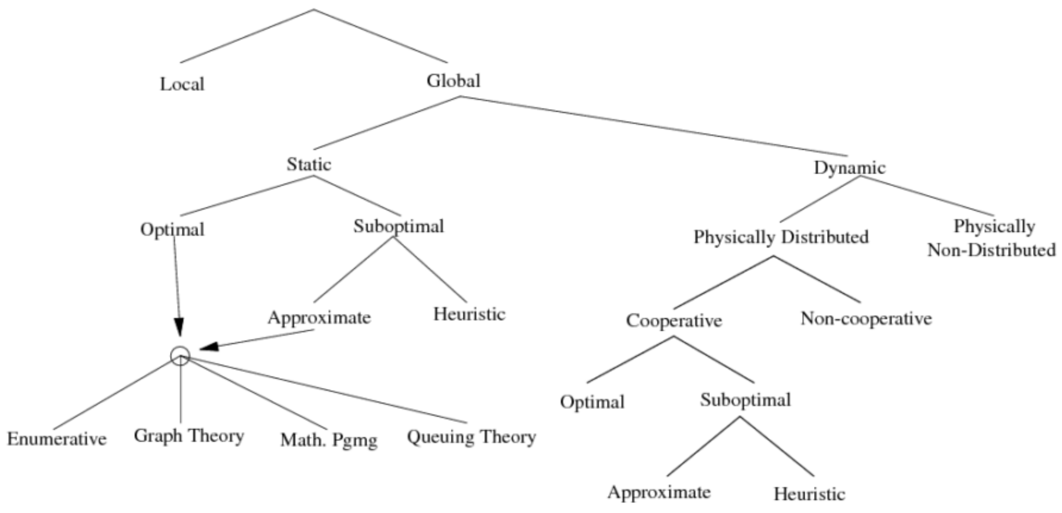

Figure 5 – Load balancing hierarchy [29]

Moreover, according to [30] in the global solution, decisions are made by a single node such as the *physically non-distributed* defined by [27].

Using other classification criteria, [31] [32] [28] [33] propose that load balancing algorithms should be divided into *static*, *dynamic* and *adaptive*. As explained in [34], *static* algorithms run a predetermined policy where the current (load) state of the system is not taken into account, unlike *dynamic* and *adaptive* algorithms. *Dynamic* algorithms where their parameters and scheduling policy may change depending of the global system state are called *adaptive* algorithms. For the sake of simplicity, this thesis adopts the classification proposed by [29] and [27].

2.3.2
Elements of a Dynamic Load Balancing Algorithm

Delving into more details of the load balancing process, a dynamic load balancing algorithm have four main elements that are: (i) *Initiation*, (ii) *Load Balancer Location*, (iii) *Information Exchange* and (iv*) Load Selection*, shown in Figure 6 [35].

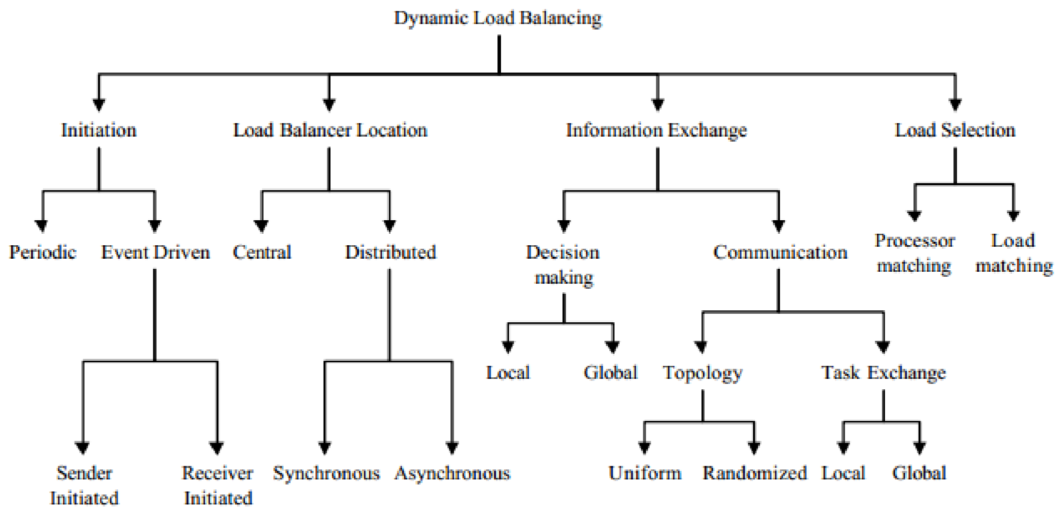

Figure 6 – Taxonomy of dynamic load balancing algorithms [35]

The *Initiation* policy defines how the current load information is exchanged among the nodes. While in a *periodic* strategy, information is exchanged at predefined time intervals, an *event-driven* initiation strategy is based on the local load observation. According to [35], the later strategy better handles load imbalance and has a lower overhead than the periodic strategy when the system load is already balanced.

A designer of load balancing algorithm should choose one of two strategies for the location of the load balancer, i.e. the node in charge of analyzing the system load and deciding whether a load redistribution among the nodes is required. Load Balancer location strategies can be *centralized* or *distributed*. Unlike the *centralized* strategy where a single node evaluates the load of the entire system, a *distributed* approach has some, or possibility all, nodes responsible for made load balancing decisions.

Because the remaining sub-strategies of the taxonomy shown in Figure 5 and Figure 6 are not of much relevance for this thesis, we refer to [35] for an in-depth discussion about the characteristics of the other policies. Moreover, the objective of this thesis is not to propose specific load distribution algorithms, but rather provide general mechanisms that support the implementation of several distributed load balancing algorithms.

2.3.3 Virtual Servers

Every load balancing solution needs a mechanism to enable the division of the whole system workload in smaller portions, thus distributing these portions over the nodes. A common load balancing technique for DHTs (Distributed Hash Tables) is the concept of *virtual servers*. A *virtual server* is similar to a logical peer (node) of the DHT. However each physical node of the system may execute one or more *virtual servers* [36] [37] [38].

The main advantage of using *virtual servers* is the possibility of splitting the load under smaller units, thus moving one (or more) *virtual server* responsible for a fixed percentage of the total load to another node [37]. However, the action of creating *virtual servers* to split the load increases the system complexity and incurs in some overhead related to their management and resource consumption since each *virtual server* acts as an independent logical node on the system [37] [38] [39]. In section 3.1 a novel technique to split the system workload is presented.

2.4 Autonomic Computing

Autonomic Computing (AC) has been inspired by the human autonomic nervous system [40] [41], which has developed strategies and algorithms to handle complexity and uncertainties. The main goal of *Autonomic Computing* is to build computing systems and applications able to manage themselves, thus minimizing human intervention [42] [40] [43] [44] [45]. According to [41] in order to accomplish the AC challenges scientific and technological advances in a wide range of fields and system architectures are required, as well as new programming paradigm and software.

Using technology to manage technology – that is software and hardware that manage themselves – requires self-management autonomic capabilities to anticipate and independently solve run-time problems [46]. Such systems should have the self-properties shown in Figure 7 [45]. Contrasting with [45], [42] mentions that most of the existing "…Self-Adaptive systems have contained some of these properties…" and gives as an example streaming media systems, which may change codec and stream quality in response to network bandwidth fluctuations. According to [43] [46] [41] the essential properties – shown in Figure 7 – to achieve the AC goal are:

- **Self-configuring:** an autonomic system must be able to dynamically adapt to changes in the environment based on policies;
- **Self-healing:** an autonomic system must be capable to detect system malfunctions and perform policy-based corrective actions without halting or disrupting the system execution;
- **Self-optimizing:** an autonomic system must have the capability to tune itself to meet the end-user/system needs or Quality of Service (QoS);
- **Self-protecting:** an autonomic system must have the capability of protect itself from accidental or malicious attacks.

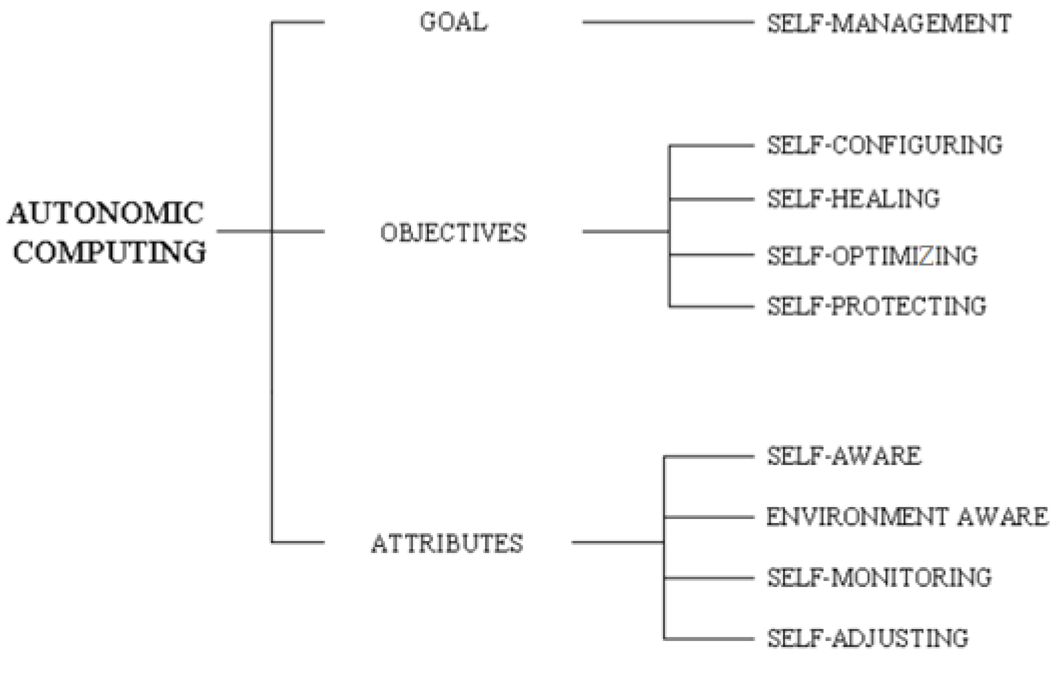

Figure 7 – General properties of Autonomic Computing [47]

In order to support the four aforementioned objectives, autonomous systems require to have self-awareness, awareness of its environment, self-monitoring functions and mechanisms for self-adjusting/-configuring, as explained in [43].

2.4.1
MAPE-K Loop

Some software architectures have been proposed to achieve the AC goal, and all these proposals include the following activities [43]:

- **Monitoring:** function that collects, aggregates, correlates and filters data about managed resources;
- **Analysis and decision:** the analysis examines data collected and determine whether changes must be made on current policies. The decision making ensures convergence according to threshold values of parameters such as performance, availability and security;
- **Control and execution:** function that performs the changes identified as necessary by the *analysis and decision* function.

Among the several proposals, the most famous one is by IBM (International Business Machines Corporation), that created the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) [46] reference model for autonomic control loops, as illustrated in Figure 8. MAPE-K is a generic autonomic control loop that abstracts characteristics of the control and data flow around its loop [48]. This reference model describes the architecture building blocks used for construct autonomic capabilities and defines a common approach and terminology for describing self-managing AC systems [46].

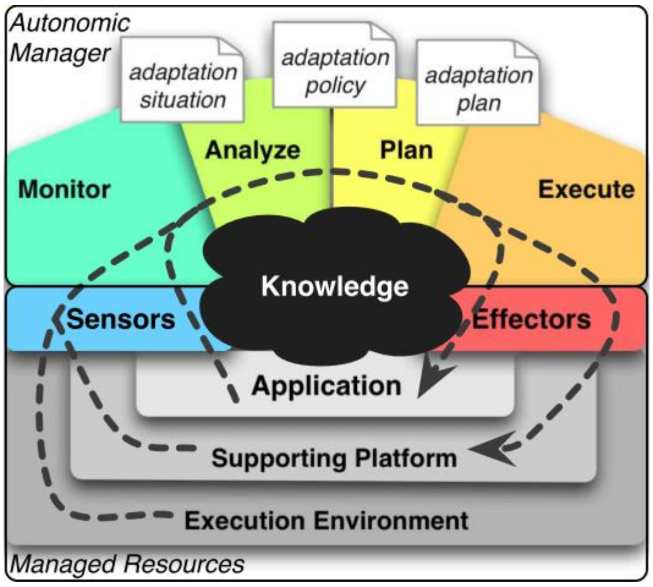

Figure 8 – MAPE-K control loop [49]

In the MAPE-K model, the *managed resources* stands for any IT system (or system component) that exhibits autonomic behavior by its tight coupling with the

MAPE-K loop. A *Sensor*, which may be implemented as a software or hardware component (e.g. CPU – Central Processing Unit – usage monitor or temperature sensor), collects information about the *managed resources* and sends this information to the *Monitor*. Also connected to the *managed resource* is the *Effector,* which is responsible for modifying the state and/or behavior of the *managed resource*. Only *sensors* and *effectors* have direct access to the *managed resource*.

The M*onitor* accesses, configures and controls a sensor for collecting raw data about the *managed resource*. This data may include details about metrics, topology system, system configuration, CPU/memory utilization and air pressure/temperature, for instance. *Monitoring* usually collects and interprets large amounts of data from the *sensors*. Therefore it should be able to aggregate, process or summarize data so as to pass consolidated data to the next MAPE-K step.

The *Analyze* function receives data from *Monitor* as its input, and is responsible for processing the data, determining whether some change needs to be made at the managed resource, and possibly generating a change request. The *Plan* is responsible for choosing and structuring the actions required to perform the changes upon the *managed resource*. In general, the *Analyze* and *Plan* functions are implemented in the same component. As the last function of MAPE-K, *Execute*, actually controls the execution of the actions generated at the previous step, Plan, by using the *Effector* to perform the actions, concluding a MAPE-K cycle.