Fault Tolerance Management in Cloud Computing: A System-Level Perspective

Ravi Jhawar, *Graduate Student Member, IEEE,* Vincenzo Piuri, *Fellow, IEEE,* and Marco Santambrogio, *Senior Member, IEEE*

***Abstract*—The increasing popularity of Cloud computing as an attractive alternative to classic information processing systems has increased the importance of its correct and continuous operation even in the presence of faulty components. In this paper, we introduce an innovative, system-level, modular perspective on creating and managing fault tolerance in Clouds. We propose a comprehensive high-level approach to shading the implementation details of the fault tolerance techniques to application developers and users by means of a dedicated service layer. In particular, the service layer allows the user to specify and apply the desired level of fault tolerance, and does not require knowledge about the fault tolerance techniques that are available in the envisioned Cloud and their implementations.**

***Index Terms*—Cloud computing, fault tolerance as a service, fault tolerance properties, system level fault tolerance.**

I. Introduction

THE INCREASING demand for flexibility in obtaining and releasing computing resources in a cost-effective manner has resulted in a wide adoption of the Cloud computing paradigm. The availability of an extensible pool of resources for the user provides an effective alternative to deploy applications with high scalability and processing requirements [23]. In general, a Cloud computing infrastructure is built by interconnecting large-scale virtualized data centers, and computing resources are delivered to the user over the Internet in the form of an on-demand service by using virtual machines (e.g., [1], [2]). While the benefits are immense, this computing paradigm has significantly changed the dimension of risks on user's applications, specifically because the failures (e.g., server overload, network congestion, hardware faults) that manifest in the data centers are outside the scope of the user's organization [3], [4]. Nevertheless, these failures impose high implications on the applications deployed in virtual machines and, as a result, there is an increasing need to address users' reliability and availability concerns.

The traditional way of achieving reliable and highly available software is to make use of fault tolerance methods at procurement and development time [26]. This implies that users must understand fault tolerance techniques and tailor their applications by considering environment-specific parameters during the design phase. However, for the applications to be deployed in the Cloud computing environment, it is difficult to design a holistic fault tolerance solution that efficiently combines the failure behavior and system architecture of the application. This difficulty arises due to: 1) high system complexity, and 2) abstraction layers of Cloud computing that release limited information about the underlying infrastructure to its users.

In contrast with the traditional approach, we advocate a new dimension where applications deployed in a Cloud computing infrastructure can obtain required fault tolerance properties from a third party. To support the new dimension, we extend our work in [5] and propose an approach to realize general fault tolerance mechanisms as independent modules such that each module can transparently function on users' applications. We then enrich each module with a set of metadata that characterize its fault tolerance properties, and use the metadata to select mechanisms that satisfy user's requirements. Furthermore, we present a scheme that: 1) delivers a comprehensive fault tolerance solution to user's applications by combining selected fault tolerance mechanisms, and 2) ascertains the properties of a fault tolerance solution by means of runtime monitoring. Based on the proposed approach, we design a framework that easily integrates with the existing Cloud infrastructure and facilitates a third party in offering fault tolerance as a service.

This paper is organized as follows. Section II describes the motivating scenario and basic concepts on fault tolerance. Section III presents our approach on resource management, and Section IV outlines our two-stage service delivery scheme that can transparently offer fault tolerance support to users' applications. Section V presents the architectural details of our framework. Section VI summarizes the related work, and Section VII presents our conclusions.

II. Motivating Scenario and Basic Concepts

In this section, we describe the motivating scenario and basic concepts on fault tolerance.

A. Motivating Scenario

We consider a highly complex and distributed infrastructure that involves the following main stakeholders.

Manuscript received October 8, 2011; revised May 8, 2012; accepted June 3, 2012. Date of publication November 29, 2012; date of current version April 17, 2013. This work was supported in part by the Italian Ministry of Research within the PRIN 2008 Project PEPPER (2008SY2PH4).

R. Jhawar and V. Piuri are with the Università degli Studi di Milano, Crema 26013, Italy (e-mail: ravi.jhawar@unimi.it; vincenzo.piuri@unimi.it).

M. Santambrogio is with the Department of Electronics and Information, Politecnico di Milano, Milan 20133, Italy (e-mail: marco.santambrogio@polimi.it).

Digital Object Identifier 10.1109/JSYST.2012.2221934

1) Infrastructure provider (IP): the entity that builds a Cloud computing infrastructure and realizes a service-oriented computing resources delivery scheme.
2) Client (C): the entity that uses the infrastructure provider's service to deploy its applications. A client meets its reliability and availability goals by making use of the service offered by the fault tolerance service provider.
3) Fault tolerance service provider (SP): the entity that provides fault tolerance support to applications based on the client's requirements. We assume that the service provider is trusted by both the infrastructure provider and the client.

As an example, we consider a client offering a web-based banking service that allows its customers to perform fund transfers and manage their accounts over the Internet. The client implements the banking service as a multitier application where: 1) the data tier uses the storage service offered by the IP to store and retrieve its customer data, and 2) the application tier uses the IP's compute service to process its operations and respond to customer queries. This system architecture allows the banking service to meet its varying business demands with respect to scalability and elasticity of computing resources. However, a failure in the IP's system can have high implications on the reliability and availability of the banking service. Furthermore, a failure in the storage server may have a significantly higher impact than a failure in one amongst several compute nodes. This implies that each tier of the banking application requires different fault tolerance properties, and the requirements may change over time based on the business demands. However, using traditional methods, fault tolerance properties of the banking service remain constant throughout its life cycle. Therefore, in the client's perspective, it is easier to engage with the SP, specify its reliability and availability requirements based on business needs, and transparently obtain desired fault tolerance properties for its applications.

B. Basic Concepts

A client engages with the service provider to obtain fault tolerance support for its applications. The goal of the service provider is to create a fault tolerance solution based on the client's requirements such that a fine balance between the following factors is achieved.

1) Fault model: measures the granularity at which the fault tolerance solution must handle errors and failures in the system. This factor is characterized by the mechanisms applied to achieve fault tolerance, robustness of failure detection protocols, and strength of fail-over granularity.
2) Resource consumption: measures the amount and cost of resources that are required to realize a fault model. This factor is normally inherent with the granularity of the failure detection and recovery mechanisms in terms of CPU, memory, bandwidth, I/O, and so on.
3) Performance: deals with the impact of the fault tolerance procedure on the end-to-end quality of service (QoS) both during failure and failure-free periods. This impact is often characterized using fault detection latency, replica launch latency and failure recovery latency, and other application-dependent metrics such as bandwidth, latency, and loss rate.

The most widely adopted strategy to tolerate failures in a system is based on the notion of redundancy. In redundancy-based schemes, critical system components are duplicated using additional hardware, software, and network resources such that a copy of critical components is available after a failure happens. For example, the data tier of the banking service can be replicated on several storage servers such that at least one copy of the data is always available to process customer queries. In general, a fault tolerance algorithm that handles failures at a finer granularity, and offers high-performance guarantees, consumes higher amount of resources. For instance, active replication methods in which all redundant components are simultaneously invoked consume more resources than passive replication methods in which only one processing node handles the requests while other replicas are simple backups. However, passive replication techniques can only handle crash faults, while active replication techniques using $3f+1$ replicas can be used to tolerate up to f arbitrary faults (e.g., [6], [7]).

We observe that a service provider must satisfy the following requirements to effectively realize its functionality and meet its business goals.

1) The service provider must always maintain a consistent view of the resources in the Cloud to efficiently deliver the fault tolerance support to its clients. To this aim, we introduce a resource manager that is maintained by the service provider (see Section III).
2) The service provider must develop: 1) an approach to realize standard fault tolerance algorithms that can extrinsically function on a client's application; 2) a method for evaluating the fault tolerance properties offered by a given mechanism and for matching it with client's requirements; and 3) a delivery scheme that can transparently enforce the desired fault tolerance properties on client applications (see Section IV).
3) The service provider must design a framework that can easily integrate with the existing Cloud infrastructure and meet service provider's goals (see Section V).

III. Resource Manager

The service provider must maintain a consistent view of all computing resources in the Cloud to efficiently allocate resources during each client request and to avoid over provisioning during failures. In this context, a resource manager that continuously monitors the working state of physical and virtual resources maintains a database of inventory and log information, and a graph representing the topology and working state of resources must be introduced by the service provider in the infrastructure provider's system.

The database of the resource manager must maintain the inventory information of each machine such as its unique serial number, composition of the machine (e.g., processor speed, number of hard disks, and memory modules), date when the machine was commissioned (or decommissioned), location of the machine in the cluster, and so on. The runtime state of

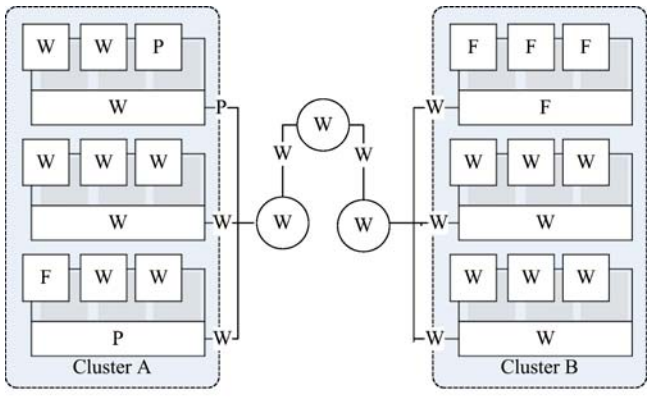

Fig. 1. Example of a graph generated by the resource manager to maintain details about all the nodes, VM instances, and network links in a Cloud. Here, *W*, *P*, and *F*, respectively, represent the working, partially faulty, and completely faulty state of a resource.

machines such as memory used/free, disk capacity used/free, and processor cores utilization must also be logged. On the other hand, a resource graph must represent the topology of resources in a system. Fig. 1 represents the resource graph $G(N, E)$ of a Cloud infrastructure where two clusters of three processing nodes each are connected by network switches. In the resource graph, each vertex represents a processing node $n \in N$, and a network link between two nodes is represented as an edge $e \in E$. A vertex also maintains information about the set of virtual machine (VM) instances hosted on that node. From the service provider's point of view, a resource graph can represent the state of nodes and edges at different granularities. In a simple case, each node and edge can be categorized in one of the three categories: working (W), partially faulty (P), and completely faulty (F). The resource manager marks the nodes (and edges) with W when they exhibit a normal condition, i.e., operational with its full potential. A node (or edge) is marked F if it has crashed or has incurred a major failure and cannot be recovered back to W. Partially faulty nodes, represented as P in the resource graph, are the ones where only a component of the node is not in use or is exhibiting a degraded performance (e.g., only the disk storage of the node is nonfunctional). Similarly, the working state of network links and VM instances must be maintained by the resource manager. We note that the database and the resource graph are essential for the service provider to ensure the correct behavior of fault tolerance mechanisms. For example, a replication mechanism may have constraints on relative placement of individual replicas and requirements on resource characteristics of each replica that can be satisfied using the resource manager [29]. We further note that the resource manager significantly contributes toward balancing the resource costs, performance, and fault model factors for the service provider.

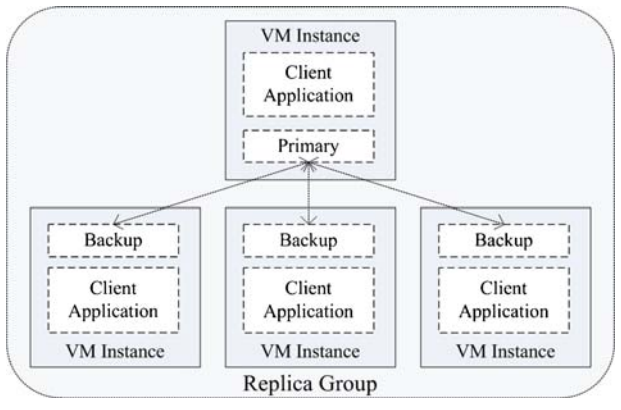

Fig. 2. Instance of an ft_unit that realizes the heartbeat-based failure detection mechanism.

IV. Fault Tolerance Delivery Scheme

The task of offering fault tolerance as a service requires the service provider to realize generic fault tolerance mechanisms such that the client's applications deployed in virtual machine instances can transparently obtain fault tolerance properties. To this aim, we define *ft_unit* as the fundamental module that applies a coherent fault tolerance mechanism to a recurrent system failure at the granularity of a VM instance. The notion of ft_unit is based on the observation that the impact of hardware failures on client's applications can be handled by applying fault tolerance mechanisms directly at the virtualization layer than the application itself (e.g., [8], [9]). For instance, fault tolerance of the banking service can be increased by replicating the entire VM instance in which its application tier is deployed on multiple physical nodes, and server crashes can be detected using well-known failure detection algorithms such as the heartbeat protocol. An example of a heartbeat protocol is depicted in Fig. 2, where the primary and backup components are run in VM instances independent of the banking service's application tier. In this example, the primary component periodically sends a liveness request to all backup components and maintains a timer for each request. When a backup receives a liveness request, it immediately responds to the primary. If the backup fails (due to a server crash) to respond to the primary for N consecutive requests, each within a predefined timeout threshold, it is suspected to failure. In this context, we note that replication of the client's application (ft_unit1), and detection of node failures (ft_unit2) are performed without requiring any changes to the application's source code. In this paper, we assume that the service provider realizes a range of fault tolerance mechanisms as ft_units, and based on this assumption we present a two-stage delivery scheme: design stage, and runtime stage, to transparently deliver high levels of fault tolerance to client's applications using ft_units.

A. Design Stage

The design stage starts when a client requests the service provider to offer fault tolerance support to its applications. In this stage, the service provider must first analyze the client's requirements, match them with available ft_units, and form a complete fault tolerance solution using appropriate

ft_units. We note that each ft_unit offers a unique set of fault tolerance properties that can be characterized using its functional, operational, and structural attributes [30]. In this context, fault tolerance property p of a ft_unit can be specified as p=(u, A), where u represents the ft_unit, and A denotes a set of attributes that refers to the granularity at which u can handle failures, the benefits and limitations of using u, inherent resource consumption costs, and QoS parameters. For each attribute $a \in A$ a partial (or total) order relationship can be defined on its domain $\mathcal{D}_a$, and $v(a)$ represents the value of a. For instance, fault tolerance property of a ft_unit u_1 can be denoted as p=(u_1, {mechanism=active_replication, no_of_replicas=4, fault_model=node_crashes}). Therefore, we propose to bind the attributes set A to the ft_unit as its metadata to facilitate a service provider in estimating fault tolerance properties that can be obtained with its use. If a client's requirements are specified in terms of expected fault tolerance properties p_c, the set S of ft_units that matches p_c can be generated by including all ft_units for which $v_i(a) \geq v_c(a)$ for each attribute $a_i \in A$ specified in p_i of u_i, i.e., all ft_units that holds the properties desired by the client. We note that there is also an implicit hierarchy of fault tolerance properties where $p_i \preceq p_j$ implies that p_j satisfies p_i. After generating the shortlisted set S, the task of the service provider is to compare each ft_unit within S and choose the one that best balances the fault model, resource costs, and performance with respect to p_c. As an example, let us consider that the service provider realizes three ft_units with properties p_1=(u_1, {mechanism=heartbeat_test, timeout_period=50 ms, N=5, fault_model=node_crashes, max_no_replicas=3}), p_2=(u_2, {mechanism=majority_voting, fault_model=programming_errors}), and p_3=(u_3, {mechanism=heartbeat_test, timeout_period=25 ms, N=3, fault_model=node_crashes, max_no_replicas=5}) respectively. If the client requests a fault tolerance support for its banking service with a more robust crash failure detection mechanism, the service provider first shortlists S=(u_1, u_3), then compares S and finally makes use of u_3 ft_unit since u_3 is more robust than u_1.

Although a ft_unit can serve as the fundamental fault tolerance module for the service provider, a comprehensive fault tolerance solution *ft_sol* that must be delivered to a client's application may be formed only by combining a set of ft_units in a specific execution logic. For example, a heartbeat test (ft_unit1) can be applied only after the client's application is replicated on multiple nodes (ft_unit2), and a recovery mechanism (ft_unit3) can be applied only after a failure is detected, that is, a comprehensive fault tolerance solution that is finally delivered to the client is as follows:

ft_sol[
`invoke`:ft_unit(*VM-instances replication*)
`invoke`:ft_unit(*failure detection*)
`do`{
`execute`(*failure detection ft_unit*)
}`while`(*no failures*)
`if`(*failure detected*)
`invoke`:ft_unit(*recovery mechanism*)
].

By using fault tolerance modules (ft_unit) to form a comprehensive solution, the dimension and intensity of the fault tolerance support can be dynamically changed. In other words, the fault tolerance properties applied on client's application can be adapted based on business needs to overcome the inflexibility of traditional fault tolerance methods. For instance, a robust failure detection mechanism (such as u_3 in the above example) can be replaced with a less robust one (u_1) in ft_sol. Furthermore, ft_units can flexibly and extensively be reused for each client request saving significant amount of resources for the service provider, and by realizing ft_units to be configurable at runtime, resource consumption costs for clients can be largely controlled. For example, by providing the parameters such as the number of replicas ($no\_of\_replicas$) for a ft_unit at runtime, the value $v(no\_of\_replicas)$=4 can be modified to $v(no\_of\_replicas)$<4 or $v(no\_of\_replicas)$>4 based on business demands. However, we note that a wide range of ft_units must be realized by the service provider to offer a higher quality of fault tolerance support that precisely meets client's requirements.

B. Runtime Stage

The runtime stage starts immediately after the service provider forms a ft_sol and delivers it on the client's application. This stage is critical for efficient service delivery since the context and attribute values of a fault tolerance solution may change at runtime due to the dynamic nature of the Cloud computing environment [25]. In other words, the mutable behavior of fault tolerance attributes requires the service provider to ascertain that the client's requirements are satisfied even during runtime. To achieve this, we first define a set $\mathcal{R}$ of rules over attributes a and their values $v(a)$ such that the validity of every rule $r \in \mathcal{R}$ establishes that property p is supported by the fault tolerance solution and violation of a rule $r_i \in \mathcal{R}$ implies that p is invalid. For instance, for a comprehensive fault tolerance solution s_1 that holds the property p_1=(s_1, {mechanism=active_replication, level=3, failure_detection=heartbeat_test, max_recovery_time=25 ms}), a set of rules $\mathcal{R}$ that can sufficiently test the validity of p_1 must be defined, such as r_1:no_of_server_instances$\geq$3, r_2:heartbeat_test_frequency=5 ms, r_3:recovery_time$\leq$25 ms. In this context, the task of the service provider is to continuously monitor the attribute values of each fault tolerance solution delivered to the client's application at runtime, and verify the corresponding set of rules $\mathcal{R}$ to ensure that client's requirements are satisfied. We note that the service provider can obtain attribute values by periodically querying each ft_sol s delivered to a client's application. Here, a fault tolerance property can be represented as p_t=(s, A_t) where t denotes the point of time at which the attribute value is queried, $v_t(a)$ is the value of attribute at t, and s is the comprehensive fault tolerance solution. We define a validation function $f(s, \mathcal{R})$ that takes s and the corresponding $\mathcal{R}$ as input and outputs *true* if $v_t(a) \geq v_i(a)$ for each attribute $a \in A$, i.e., $f(s, \mathcal{R})$ verifies whether the fault tolerance solution remains valid and satisfies the client's requirements at runtime. In case, $f(s, \mathcal{R})$ returns *false*, the service provider must trigger the matching and comparison process of the design phase to select a new set

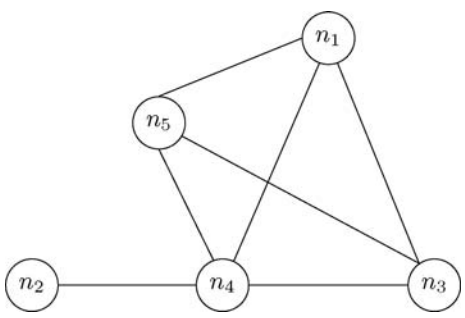

Fig. 3. Instance of a resource graph generated by the resource manager.

of ft_units and form a new ft_sol that best matches client's requirements. Therefore, by constantly monitoring each ft_sol and by updating the attribute values, the service provider can deliver a fault tolerance support that is valid throughout the life cycle of the client's application (initially during request time and at runtime). Furthermore, a change in the client's requirements at any stage also triggers the design phase to form a new fault tolerance solution.

V. Fault Tolerance Manager: Architecture Framework

In this section, we present a conceptual framework, the Fault Tolerance Manager (FTM), that provides the basis for a service provider to realize the delivery scheme presented in the previous section and hence to offer fault tolerance as a service. We aim to insert our framework as a dedicated service layer between the client's applications and the hardware that works directly on the top of the virtual machine manager at the level of VM instances. The Fault Tolerance Manager must address the issue of heterogeneity in computing resources, fulfill the target of transparently providing fault tolerance support to user's applications against node failures, and satisfy scalability and interoperability goals. To overcome these challenges, we propose to build the Fault Tolerance Manager using the principles of service-oriented architecture, where each ft_unit is realized as an individual web service, and a ft_sol is formed using the business process execution language (BPEL) constructs [10], [11]. We include a resource manager within FTM that initially coordinates with the cloud manager to produce the resource graph and the database, as discussed in Section III. The resource manager is realized in the form of a web service that provides a status operation that takes a resource (e.g., processing node, storage, memory) as input and outputs the state of that resource. Note that status operation can be run independently on each node and, using the `update` operation, state of the resource can be updated in the database and resource graph. As an example, let us consider that at the start of the service invocation, the service provider generates a profile of computing resources in the cloud infrastructure by identifying five processing nodes $\{n_1, \ldots, n_5\} \in N$ whose resource graph is presented in Fig. 3. A description of all the components in the framework is provided further in this section.

A. Client Interface

The service invocation process begins when a client requests the service provider to offer fault tolerance support to its applications with a desired set of properties. In this context, it is essential to include a client interface component within FTM that provides a specification language that allows clients to specify and define their requirements (e.g., [12], [13], [31]). However, since the present-day cloud computing systems require its users to manage their VMs while dealing with sophisticated system-level concerns, an automated configuration tool that requires clients to simply select the application for which they wish to obtain fault tolerance support, and correspondingly provide values of desired availability, reliability, response time, criticality of the application and cost can be beneficial. We note that an automated configuration tool can limit human errors and save time by lessening the need for manual tedious configuration. Moreover, if the input can be provided in a high-level format (such as percentages, range, and numbers), even users with a nontechnical background can configure the desired properties with ease. We consider the aspect of transforming high-level metric values into a set of fault tolerance properties, and translating the properties in terms of standard fault tolerance mechanisms as part of our future work.

B. FTMKernel

The central computing component of our framework is the FTMKernel that is responsible for composing a fault tolerance solution based on client's requirements using the web service modules (ft_units) implemented by the service provider, delivering the composed service on client's applications, and monitoring each service instance to ensure its QoS. FTMKernel is composed of a service directory, a composition engine, and an evaluation unit.

1) Service directory: It is a registry of all ft_units realized by the service provider in the form of web services. A ft_unit applies a fault tolerance mechanism as a self-contained loosely coupled module, with a well-defined language-agnostic interface that: 1) describes its operations and input or output data structures (e.g., WSDL and WSCL), and 2) allows other ft_units to coordinate and assemble with it. In addition to the ft_units, this component also registers the metadata that represents the fault tolerance property p=(u, A) of each ft_unit. When FTM receives input from the client interface, this component first performs a matching between the client's preferences p_c, and properties p_i of each ft_unit in the service directory, to generate the set S of ft_units that satisfy p_c. The set S is then ordered based on client's preferences and provided to the composition engine. The service directory triggers the matching and comparison processes at runtime if the evaluation unit updates the metadata of a ft_unit. However, we note that the service provider must perform an *a priori* validation of all its ft_units and estimate their properties p in the infrastructure provider's system as a prerequisite.
2) Composition engine: It receives an ordered set of ft_units from the service directory as an input, and

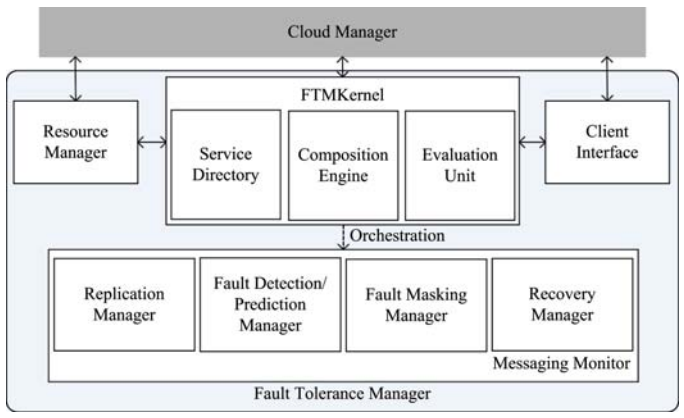

Fig. 4. Architectural overview of FTM showing various components.

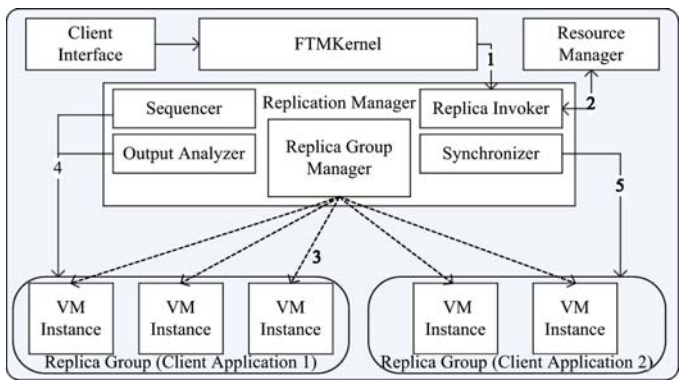

Fig. 5. Overview of various components in replication manager, and their interaction with other components of the framework. The straight lines with numbers describe the replication process, and the operations at each step are correspondingly explained in Section V-C. The dotted lines represent the interaction with the VM instances that are external to the framework.

generates a comprehensive fault tolerance solution ft_sol using the web services (ft_units) that best match client's preferences as an output. In terms of service-oriented architecture, the composition engine can be viewed as a web service orchestration engine that exploits BPEL constructs to build a composed fault tolerance solution that is delivered to client's applications using robust message exchanges protocols (e.g., [14]), as presented in Section IV.

3) Evaluation unit: It continuously monitors all composed fault tolerance solutions at runtime using the validation function $f(s, \mathcal{R})$ and the set of rules $\mathcal{R}$ defined corresponding to each ft_sol. We note that the interface exposed by web services (e.g., WSDL and WSCL) allows the evaluation unit to validate all the rules $r \in \mathcal{R}$ during runtime monitoring. If $f(s, \mathcal{R})$ returns *false*, the evaluation unit updates the present attribute values in the metadata; otherwise, the service continues uninterrupted.

FTMKernel can measure the overall reliability of the service provided to the client's applications by comparing a set of metrics [such as mean time between failure (MTBF)] between the real-time operational data obtained from the resource database, and expected values of the metrics obtained from the input using the client interface. For example, a client's request for 99% availability of its applications implies that FTM must ensure that the MTBF of the node where the application is deployed is at least $avail_{exp} = 0.99 * t$ for a given time period t. Since each node failure is logged in the database, the operational $avail_{real}$ value can be calculated, and the strength of the service provided by FTM by measuring $avail_{exp} - avail_{real}$.

In addition to the resource manager, client interface, and FTMKernel, we note that our framework must include a set of components that provide a complementary support to fault tolerance mechanisms. These components significantly affect the quality of service offered by the service provider, and are essential to satisfy client's requirements and constraints. We include the following components in our framework, and present the overall architecture of the Fault Tolerance Manager in Fig. 4.

1) Replication Manager: provides support to ft_units that realize replication mechanisms by managing the details regarding individual replicas of a client's application, their location, and synchronization process between them (see Section V-C).
2) Fault Detection or Prediction Manager: provides support to techniques that either detect or predict failures among the nodes (see Section V-D).
3) Fault Masking Manager: comprise modules that support techniques that are used to mask the presence of faults in the system (see Section V-E).
4) Recovery Manager: includes services that support ft_units that recovers a faulty node back to operational (see Section V-F).
5) Messaging Monitor: provides the infrastructure necessary for communication among all the components of FTM (see Section V-G).

In the following sections, we provide a detailed description of these components.

C. Replication Manager

This component supports the replication mechanisms by invoking replicas and managing their execution based on the client's requirements. We denote the set of VM instances that are controlled by a single implementation of a replication mechanism (ft_unit) as a replica group. Each replica within a group can be uniquely identified, and a set of rules $\mathcal{R}$ that must be satisfied by a replica group are specified. The task of the replication manager is to make the client perceive a replica group as a single service, and to ensure that the fault free replicas exhibit correct behavior during execution time.

Fig. 5 provides an overview of various components within the replication manager and their interactions with each other. To support a replication mechanism, the replica invoker first contemplates the desired replication parameters such as the style of replication (active, passive, cold passive, hot passive), number of replicas, and constraints on relative placement of individual replicas, and forms the replica group. In other words, the replica invoker takes the reference of a client's application as input from FTMKernel, analyzes the expected fault tolerance properties, and interacts with the resource manager to obtain the location of each replica. The replica

group manager then creates the replica group by invoking VM instances at those locations and managing their execution. The sequencer provides the input to application executing in the replica group by means of consensus protocol (e.g., [7], [15]) in order to ensure determinism among replicas. The output analyzer carries out majority voting on the responses obtained, and returns the chosen result to the client. The synchronizer includes techniques to update the state of backup replicas with that of the primary in a replica group. It also supports membership change and primary election algorithms when the primary node undergoes failure. We note that robustness of these procedures largely contribute to the consistency and reliability of the service.

Example: Let us consider that FTMKernel chooses a passive replication mechanism corresponding to the banking service's request where the following constraints must be satisfied: 1) the replica group must contain one primary and two backup nodes at all times; 2) the node on which the primary executes must not be shared with any other VM instances; and 3) all the replicas must be located on different nodes. For the Cloud infrastructure depicted in Fig. 3, the replication manager forms a replica group of the banking service's application by choosing the node n_1 for the primary, and nodes n_3 and n_4, respectively, for backup replicas. We note that n_3 and n_4 can host VM instances of other replica groups, while only one VM instance can run on n_1. The synchronizer of replication manager frequently checkpoints the primary and updates the state of backup replicas.

D. Fault Detection Or Prediction Manager

This component enriches the FTM by providing failure detection support at two different levels. The first level is infrastructure-centric, and provides failure detection globally across all the nodes in the Cloud, whereas the second level is application-centric and provides support only to detect failures among individual replicas within a replica group. To realize failure detection at either levels, we note that this component must support several well-known failure detection algorithms (e.g., the gossip based protocol, and heartbeat protocol) that are configured at runtime based on replication mechanisms and client's requirements.

When the replication manager successfully creates a replica group, the composition engine invokes ft_units to detect or predict failures within the replica group. An example of a failure detection ft_unit (heartbeat protocol) is presented in Section IV. The main goal of the failure detection or prediction manager is to support FTM in detecting faults immediately after their occurrence, and sending a notification about the faulty replica to the fault masking manager and the recovery manager. For infrastructure-centric failure detection, failure notifications are sent to the resource manager to update the resource state of the cloud that is utilized to predict failures in a proactive fault tolerance approach. We note that most failure detection protocols that are exploited in a passively replicated system perform well in detecting major failures. However, to detect errors at smaller granularity resulting from a partially faulty node, active replication methods need to be deployed. For example, programming errors in client's application can be detected by applying a majority voting using the output analyzer of the replication manager on the output generated by each active replica.

Example: For the replica group of the banking service's application described in the example of Section V-C, suppose that the service directory selects a ft_unit that realizes a proactive fault tolerance mechanism. This implies that the failure detection or prediction manager must continuously gather the state information of nodes n_1, n_3, and n_4, and verify if all system parameter values are over a certain threshold (e.g., physical memory usage of a node allocated to a VM instance must be less than 70% of its total capacity).

E. Fault Masking Manager

The goal of this component is to support ft_units that realize fault masking mechanisms so that occurrence of faults in the system can be hidden from clients. When a failure is detected in the system, this component immediately applies masking procedures to prevent faults from resulting into errors. We note that the functionality of this component is critical to meet client's high availability requirements.

Example: From the example in Section V-D, let us consider that the failure detection or prediction manager predicts a failure in node n_3 and immediately invokes the fault masking ft_unit. Here, the ft_unit performs a live migration of the VM instance (e.g., [8], [9]) such that the entire OS at node n_3 is moved to another location (node n_5) while maintaining the established session so that customers of the banking service do not experience any impact of the failure at node n_3. Therefore, client's high-availability requirements can be fulfilled using the fault masking mechanisms.

F. Recovery Manager

The goal of this component is to achieve system-level resilience by minimizing the downtime of the system during failures. To this aim, this component supports ft_units that realize recovery mechanisms so that an error-prone node can be resumed back to a normal operational mode. In other words, this component provides support that is complementary to that of the failure detection or prediction manager and fault masking manager, especially in the condition when an error is detected in the system. We note that FTM maximizes the life-time of the Cloud infrastructure by continuously checking for occurrence of faults using the failure detection or prediction manager and, when exceptions happen, by recovering from failures using the recovery manager.

Example: As described in the example of Section V-E, using the fault masking manager the high-availability goals of the client can be met even when a failure happens at node n_3. However, the service offered by the infrastructure provider may be affected since the system consists of only four working nodes. In this context, it is critical for the infrastructure provider to apply robust recovery mechanisms in order to increase its system's lifetime. The support offered by the recovery manager resumes node n_3 (that is marked with F or P in the resource graph) to working state (W in a resource graph).

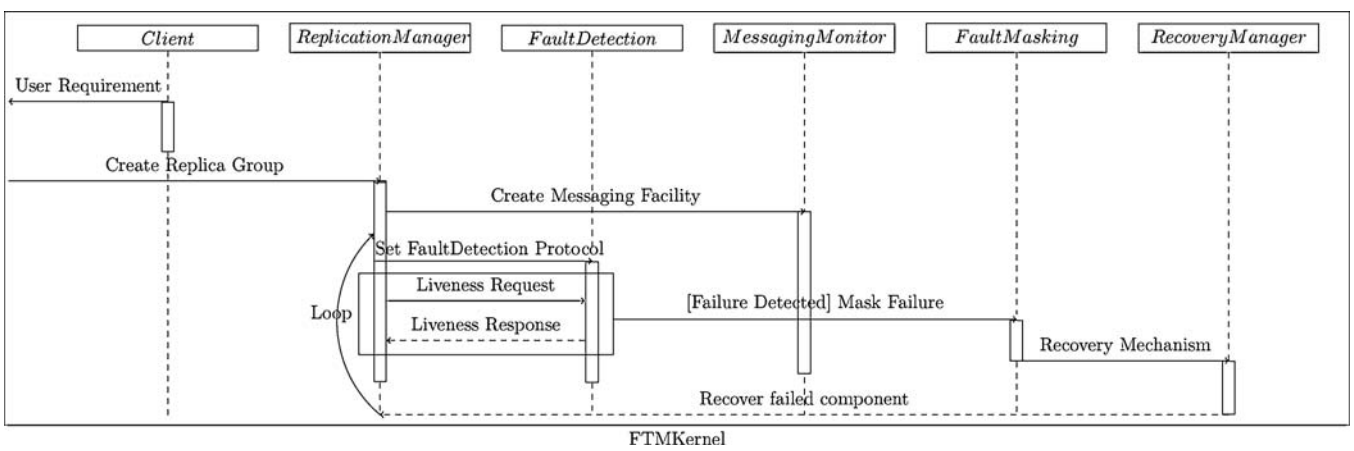

Fig. 6. Example of workflow represented as a sequence diagram showing the interaction among all the components of FTM for a single client request.

G. Messaging Monitor

The messaging monitor extends through all the components of our framework (as shown in Fig. 4) and offers the communication infrastructure in two different forms: message exchange within a replica group, and intercomponent communication within the framework. Since ft_units, and other components in FTM, are realized as web services, the communication between any two components (and the replicas) must be reliable even in the presence of component, system or network failure. To this aim, the messaging monitor integrates WS-RM standard [14], [16] with other application protocols and establishes an appropriate messaging infrastructure that supports the composition engine in designing a robust ft_sol. We note that this component is critical in providing maximum interoperability, and serves as a key QoS factor.

Example: Based on the examples presented in Sections V-C to V-F, here, we summarize the implicit workflow that happens across all the components of FTM as a response to a single-client request. As shown in Fig. 6, the service invocation process starts when FTMKernel gathers the client's requirements from the client interface using the $<receive>$ BPEL activity. Based on client's requirements, FTMKernel first selects appropriate web service modules (ft_units) from the service directory, and the composition engine defines (ft_sol) an execution logic among selected web service modules. FTMKernel delivers the ft_sol by first triggering the replication manager to create a replica group for the client's application. In terms of BPEL language constructs, activity $<invoke>$ is specified. Once the replica group is created, FTMKernel triggers the messaging monitor and the failure detection/prediction manager to create a messaging infrastructure and to invoke fault prediction protocol respectively, using the $<flow>$ activity so that both ft_units run continuously in parallel. The ft_unit associated with the fault masking manager is triggered immediately ($<if>$ activity) after a failure is predicted, and finally the recovery manager is invoked (using $<invoke>$ activity) to recover the failed node. Note that throughout the workflow, the evaluation unit monitors the service instance to ascertain that FTM satisfies the client's requirements and maintains the QoS.

VI. Related Work

A large number of fault tolerance techniques that are closely integrated with distributed software applications during development time have been proposed (e.g., [7], [15], [17]). In [6], the authors presented a fault tolerance middleware that uses the leader or follower replication approach to tolerate crash faults in the Cloud computing environment.

An interesting line of research relevant to this paper proposes an approach to build fault tolerance protocols by composing microprotocols and combining them into a system using hierarchical techniques [18]. This approach of implementing fault tolerance protocols demonstrates benefits in terms of easy customization when compared to a monolithic system. In [19], the authors used a modular approach to develop a proactive fault tolerance framework that can tailor a requirement-specific strategy, and demonstrated it to be a good platform for the study of various proactive fault tolerance policies. In this paper, we enhance individual fault tolerance modules with metadata to analyze its characteristics, and present an approach to generically compose each module so as to form a comprehensive solution with specific fault tolerance properties.

The aspect of leveraging virtualization to improve availability and reliability of the system has been used in the past. In [8] and [9], the authors presented a mechanism to continuously synchronize the memory state of a node to backup nodes using checkpointing. When the primary node failure happens, the backup node resumes execution immediately, providing semblance of no interruption to the end user. In this paper, we make use of the virtualization layer to transparently introduce fault tolerance on deployed applications.

Our framework is designed using the principles of service-oriented architecture and makes use of several web service standards, such as the WS-RM and WS-BPEL [11], [14]. In this direction, several open-source projects, such as the Sandesha2 that provides a messaging infrastructures for Apache Axis2, are available in the market [16], [20]. The authors [21] extended Sandesha2 to implement the well-known Castro and Liskov's Byzantine fault tolerance algorithm [7], and showed that the extension introduced only a moder-

ate runtime overhead. Therefore, the functionality of FTM can be efficiently implemented by extending the available tools.

VII. Conclusion and Future work

We presented an approach toward transparently delivering fault tolerance on the applications deployed in virtual machine instances. In particular, we presented an approach for realizing generic fault tolerance mechanisms as independent modules, validating fault tolerance properties of each mechanism, and matching user's requirements with available fault tolerance modules to obtain a comprehensive solution with desired properties. The proposed approach when combined with our delivery scheme enables a service provider to offer long-standing fault tolerance support to client's applications. Furthermore, we designed a framework that allows the service provider to integrate its system with the existing Cloud infrastructure and provides the basis to generically realize our approach in delivering fault tolerance as a service. The components of our framework can be extended to improve the overall resilience of the Cloud infrastructure. Our future work will mainly be driven toward the implementation of the framework to measure the strength of fault tolerance service and to make an in-depth analysis of the cost benefits among all the stakeholders.

References

[1] *Amazon Elastic Compute Cloud* [Online]. Available: http://aws.amazon.com/ec2/

[2] Eucalyptus Systems [Online]. Available: http://www.eucalyptus.com/

[3] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 193–204.

[4] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures Comput. Architecture*, vol. 4, no. 1, pp. 1–108, 2009.

[5] R. Jhawar, V. Piuri, and M. D. Santambrogio, "A comprehensive conceptual system-level approach to fault tolerance in cloud computing," in *Proc. IEEE Int. Syst. Conf.*, Mar. 2012, pp. 1–5.

[6] W. Zhao, P. M. Melliar-Smith, and L. E. Moser, "Fault tolerance middleware for cloud computing," in *Proc. 3rd Int. Conf. Cloud Comput.*, Jul. 2010, pp. 67–74.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd Symp. Operating Syst. Design Implementation*, 1999, pp. 173–186.

[8] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proc. 5th USENIX Symp. Networked Syst. Design Implementation*, 2008, pp. 161–174.

[9] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," in *Proc. USENIX Annu. Tech. Conf. (Poster Session)*, 2008.

[10] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Englewood Cliffs, NJ: Prentice-Hall PTR, 2005.

[11] *OASIS Web Services Business Process Execution Language Version 2.0 (WS-BPEL)* [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[12] Y. Mao, C. Liu, J. E. van der Merwe, and M. Fernandez, "Cloud resource orchestration: A data-centric approach," in *Proc. 5th Biennial Conf. Innovative Data Syst. Res.*, 2011, pp. 241–248.

[13] G. Koslovski, W.-L. Yeow, C. Westphal, T. T. Huu, J. Montagnat, and P. Vicat-Blanc, "Reliability support in virtual infrastructures," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, Nov. 2010, pp. 49–58.

[14] *OASIS Web Services Reliable Messaging (WSRM)* [Online]. Available: http://www.oasis-open.org/committees/tc-home.php?wg-abbrev=wsrm

[15] P. Narasimhan, K. Kihlstrom, L. Moser, and P. Melliar-Smith, "Providing support for survivable CORBA applications with the immune system," in *Proc. 19th IEEE Int. Conf. Distributed Comput. Syst.*, May 1999, pp. 507–516.

[16] *Apache axis2/java* [Online]. Available: http://axis.apache.org/axis2/java/core/

[17] N. Ayari, D. Barbaron, L. Lefevre, and P. Primet, "Fault tolerance for highly available internet services: Concepts, approaches, and issues," *IEEE Commun. Surveys Tutorials*, vol. 10, no. 2, pp. 34–46, Apr.–Jun. 2008.

[18] M. Hiltunen and R. Schlichting, "An approach to constructing modular fault-tolerant protocols," in *Proc. 12th Symp. Reliable Distributed Syst.*, 1993, pp. 105–114.

[19] G. Vallee, K. Charoenpornwattana, C. Engelmann, A. Tikotekar, C. Leangsuksun, T. Naughton, and S. L. Scott, "A framework for proactive fault tolerance," in *Proc. 3rd Int. Conf. Availability Reliability Security*, Mar. 2008, pp. 659–664.

[20] *Apache Sandesha2* [Online]. Available: http://axis.apache.org/axis2/java/sandesha/

[21] W. Zhao, "Bft-ws: A Byzantine fault tolerance framework for web services," in *Proc. 11th Int. Enterprise Distributed Object Comput. Conf. Workshop*, Oct. 2007, pp. 89–96.

[22] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," in *Proc. USENIX Annu. Tech. Conf. (Poster Session)*, 2008.

[23] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," EECS Dept., Univ. California, Berkeley, UCB/EECS-2009-28, Feb. 2009.

[24] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati, "Encryption-based policy enforcement for cloud storage," in *Proc. 30th Int. Conf. Distributed Comput. Syst. Workshop*, 2010, pp. 42–51.

[25] P. Samarati and S. De Capitani di Vimercati, "Data protection in outsourcing scenarios: Issues and directions," in *Proc. 5th ACM Symp. Inform. Comput. Commun. Security*, 2010, pp. 1–14.

[26] F. Distante and V. Piuri, "Hill-climbing heuristics for optimal hardware dimensioning and software allocation in fault-tolerant distributed systems," *IEEE Trans. Reliab.*, vol. 38, no. 1, pp. 28–39, Apr. 1989.

[27] V. Piuri, "Design of fault-tolerant distributed control systems," *IEEE Trans. Instrum. Meas.*, vol. 43, no. 2, pp. 257–264, 1994.

[28] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House, 2001.

[29] R. Guerraoui and M. Yabandeh, "Independent faults in the cloud," in *Proc. 4th Int. Workshop Large Scale Distributed Syst, Middleware*, no. 6. 2010, pp. 12–17.

[30] C. A. Ardagna, E. Damiani, R. Jhawar, and V. Piuri, "A model-based approach to reliability certification of services," in *Proc. 6th IEEE Int. Conf. Digit. Ecosyst. Technol.*, Jun. 2012, pp. 1–6.

[31] A. Heddaya and A. Helal, Reliability, Availability, Dependability and Performability: A User-centered View, Boston University, Boston, MA, 1997.

[32] S. S. Kulkarni and K. N. Biyani and U. Arumugam, "Composing distributed fault-tolerance components," in *Proc. Int. Conf. Dependable Syst. Netw., Supplemental Volume, Workshop Principles Dependable Syst.*, Jun. 2003, pp. W127–W136.

Ravi Jhawar (GSM'12) received the B.Tech. degree in computer science and engineering from Amrita Vishwa Vidyapeetham University, India, in 2009, and the M.Sc. degree in information security from University College London, London, U.K., in 2010. He is currently pursuing the Ph.D. degree with the Doctoral School of Computing Science, Università degli Studi di Milano, Crema, Italy.

He is currently with the Department of Computer Science, Università degli Studi di Milano. His current research interests include system dependability, with a focus on information security, reliability, and availability by means of fault tolerance in the Cloud computing scenario.

Vincenzo Piuri (S'86–M'96–SM'00–F'01) received the Ph.D. degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 1989.

He was an Associate Professor with the Politecnico di Milano, from 1992 to 2000, and a Visiting Professor with the University of Texas, Austin, from 1996 to 1999. He has been a Full Professor of computer engineering since 2000 and is the Director of the Department of Information Technology, Università degli Studi di Milano, Crema, Italy, from 2007 to 2012. He has participated in several national and international research projects funded by the European Union, the Italian Ministry of Research, the National Research Council of Italy, the Italian Space Agency, and various industries. His original results have been published in more than 350 papers in international journals, proceedings of international conferences, books, and book chapters. His current research interests include fault tolerance, digital processing architectures, embedded systems, cryptographic architectures, arithmetic architectures, biometrics, signal and image processing, pattern analysis and recognition, theory and industrial applications of neural networks, machine learning, intelligent measurement systems, and industrial applications.

Dr. Piuri is a Distinguished Scientist of the ACM and a Senior Member of INNS. He has been an Associate Editor of the IEEE TRANSACTIONS ON NEURAL NETWORKS and the IEEE TRANSACTIONS ON INSTRUMENTATION AND MEASUREMENT. He has been the President of the IEEE Computational Intelligence Society, the Vice President for Publications of the IEEE Instrumentation and Measurement Society and the IEEE Systems Council, the Vice President for Membership of the IEEE Computational Intelligence Society, and the Vice President for Education of the IEEE Biometrics Council. He was the IEEE Director from 2010 to 2012. He received the IEEE Instrumentation and Measurement Society Technical Award in 2002 for contributions to the advancement of theory and practice of computational intelligence in measurement systems and industrial applications, the IEEE Instrumentation and Measurement Society Distinguished Service Award in 2008, and the IEEE Computational Intelligence Society Meritorious Service Award in 2009.

Marco D. Santambrogio (M'05–SM'11) received the Laurea (M.Sc. equivalent) degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2004, the second M.Sc. degree in computer science from the University of Illinois, Chicago (UIC), in 2005, and the Ph.D. degree in computer engineering from the Politecnico di Milano, in 2008.

He is currently an Assistant Professor with the Department of Electronics and Information, Politecnico di Milano, a Research Affiliate with the Massachusetts Institute of Technology (MIT), Cambridge, and an Adjunct Professor with UIC. He was with the Computer Science and Artificial Intelligence Laboratory, MIT, as a Post-Doctoral Fellow. He was with the Micro Architectures Laboratory, Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design Project in 2004 and the CHANGE Project (Self-Aware and Adaptive Computing Systems) in 2010. He has conducted research and taught in the areas of reconfigurable computing, computer architectures, operating systems, hardware or software codesign, embedded systems, and high-performance processors and systems.

Dr. Santambrogio is a member of the IEEE Computer Society and the IEEE Circuits and Systems Society.