

Fault-Aware Application Management Protocols

Antonio Brogi, Andrea Canciani, Jacopo Soldani

▶ **To cite this version:**

Antonio Brogi, Andrea Canciani, Jacopo Soldani. Fault-Aware Application Management Protocols. 5th European Conference on Service-Oriented and Cloud Computing (ESOCC), Sep 2016, Vienna, Austria. pp.219-234, 10.1007/978-3-319-44482-6_14 . hal-01638591

HAL Id: hal-01638591
https://inria.hal.science/hal-01638591

Submitted on 20 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Fault-aware application management protocols

Antonio Brogi, Andrea Canciani, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract. We introduce *fault-aware management protocols*, which permit modelling the management behaviour of application components by taking into account the potential occurrence of faults, and we show how such protocols can be composed to analyse the behaviour of a multi-component application and to automate its management. We also illustrate a way to recover applications that are stuck because a fault was not properly handled and/or because a component is behaving differently than expected.

Keywords: *Management protocols*, *fault modelling*, *finite state machines.*

1 Introduction

Automating the management of composite applications is currently one of the major concerns of enterprise IT [18]. Such applications typically integrate various heterogeneous components, whose deployment, configuration, enactment, and termination must be suitably coordinated.

A convenient way to represent the structure of a composite application is a topology graph [3], whose nodes represent the application components, and whose arcs represents the dependencies among such components. More precisely, each topology node can be associated with the requirements of a component, the operations to manage it, and the capabilities it features. Inter-node dependencies associate the requirements of a node with capabilities featured by other nodes.

In [4] we have shown how the management behaviour of topology nodes can be modelled by *management protocols*, specified as finite state machines whose states and transitions are associated with conditions defining the consistency of the states of a node and constraining the executability of management operations. Such conditions are defined on the requirements of a node, and each requirement of a node has to be fulfilled by a capability of another node. As a consequence, the management behaviour of a composite application can be easily derived by composing the management protocols of its nodes according to the dependencies defined in its topology.

[4] does not deal with deal with the potential occurrence of faults, which however must be considered when managing complex composite applications [9]. Indeed, an application component may be affected by faults caused by other components on which it relies (e.g., a component is shutdown or uninstalled while another component is relying on its capabilities).

In this paper we propose a *fault-aware* extension of management protocols, to permit modelling how nodes behave when faults occurs. We also illustrate

how to analyse and automate the management of composite applications in a fault-resilient manner. Namely, we show how the fault-aware management behaviour of a composite application can be determined by simply composing the management protocols of its nodes according to the application's topology. We then describe how to determine whether a plan orchestrating the application management is valid, which are its effects (e.g., which capabilities are available after executing it, or whether it may generate faults while being executed), and how this also permits finding management plans from given application configurations to achieve specific goals.

Even if application components are described by fault-aware management protocols, the actual behaviour of components may differ from their described behaviour (e.g., because of some bug). We show how the unexpected behaviour of a component can be modelled by automatically completing its management protocol, and how this permits analysing the (worst possible) effects of a misbehaving component on the rest of an application. We also illustrate a way to recover applications that are stuck because a fault was not properly handled and/or because of misbehaving components.

The rest of the paper is organised as follows. Sect. 2 provides an example motivating the need for fault-aware management protocols. Sect. 3 illustrates such protocols and to compose them to analyse an application's management in presence of faults. Sect. 4 describes how to deal with faults caused by the unexpected behaviour of component(s), and how to recover stuck applications. Sects. 5 and 6 discuss related work and draw some concluding remarks.

2 Motivating example

Consider a toy application composed by a web-based front-end and a back-end, both deployed on an apache server, which in turn is installed on a debian operating system. Fig. 1 illustrates the topology of such application, according to the TOSCA [21] graphical notation.

Each inter-node dependency is explicitly represented by a relationship connecting a node's requirement with another node's capability (e.g., the server requirements of front-end and back-end are connected with the app-rte capability of apache). A relationship can represent a *vertical* containment dependency

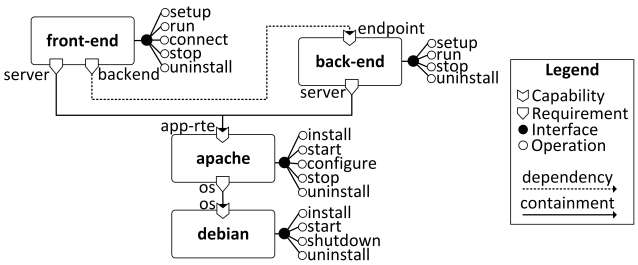


Fig. 1. Motivating example.

(e.g., apache is installed on debian), or an *horizontal* dependency, specifying that a component requires another, without stating that the former is contained in the latter (e.g., front-end must connect to back-end's endpoint to work properly).

Suppose for instance that all nodes have been deployed, started, and properly connected each other (i.e., all components are in their *running* state). What happens if the stop operation of back-end is executed? The back-end application component is stopped, and this generates a fault in the front-end, which becomes unable to serve requests to its clients, simply because the connection dependency with back-end is not working any more. Furthermore, even if back-end is re-started, the front-end has to re-connect to the back-end.

Even worse is the case when a node presents an unexpected behaviour. Suppose again that the application is up and running, and that the apache server unexpectedly crashes. Such a crash results in faulting also the nodes contained in apache (viz., front-end and back-end), which are suddenly killed, and potentially enter in an inconsistent state that makes them unusable from there onwards.

Both the above mentioned cases fail because a node stops providing its capabilities while other nodes are relying on them to continue to work. In the first case this happens because of the invocation of a management operation that stops a node while other nodes are depending on it. In the second case a node unpredictably fails[1].

3 Modelling and analysing application management in presence of faults

3.1 Fault-aware management protocols

Let N be a node modelling an application component. Management protocols [4] permit modelling the management behaviour of N by describing whether and how the management operations of N depend (i) on other operations of the same node, and/or (ii) on operations of other nodes providing the capabilities that satisfy the requirements of N.

The first kind of dependencies is described by specifying relationships between states and management operations of N. More precisely, to describe the order in which the operations of N can be executed, we employ a transition relation τ specifying whether an operation o can be executed in a state s, and which state is reached by executing o in s.

The second kind of dependencies is described by associating transitions and states with (possibly empty) sets of requirements to indicate that the corresponding capabilities are assumed to be provided. More precisely, the requirements associated with a transition t specify which are the capabilities that must be offered to allow the execution of t. Instead, the requirements associated with a state of N specify which capabilities must (continue to) be offered by other

[1] Misbehaving components can be detected via monitoring (e.g., by exploiting watchdogs or heartbeat services). We shall not deepen into details, as component monitoring is outside of the scope of this paper.

nodes in order for N to (continue to) work properly. To complete the description, each state s of N is also associated to the capabilities provided by N in s.

We hereby define a *fault-aware* extension of *management protocols* [4], to permit describing how N reacts when it is in a state assuming some requirements to be satisfied, and some other node(s) stop(s) providing the capabilities satisfying such requirements. We introduce a new transition relation φ to model the explicit fault handling of N, i.e. how N changes its state from s to s' when some of the requirements it assumes in s stop being satisfied.

Definition 1. *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where S_N, R_N, C_N, and O_N are the finite sets of its states, requirements, capabilities, and management operations, and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is a finite state machine defining the* fault-aware management protocol *of N, where*[2]*:*

- *$\overline{s}_N \in S_N$ is the initial state,*
- *$\rho_N : S_N \to 2^{R_N}$ is a function indicating which requirements must hold in each state $s \in S_N$,*
- *$\chi_N : S_N \to 2^{C_N}$ is a function indicating which capabilities of N are offered in a state $s \in S_N$,*
- *$\tau_N \subseteq S_N \times 2^{R_N} \times O_N \times S_N$ is a set of quadruples modelling the transition relation, i.e. $\langle s, H, o, s' \rangle \in \tau_N$ denotes that in state s, and if the requirements in H are satisfied, o is executable and leads to state s', and*
- *$\varphi_N \subseteq S_N \times 2^{R_N} \times S_N$ is a set of triples modelling the explicit fault handling for a node, i.e. $\langle s, F, s' \rangle \in \varphi_N$ denotes that the node will change its state from s to s' if the requirements in F stop being satisfied.*

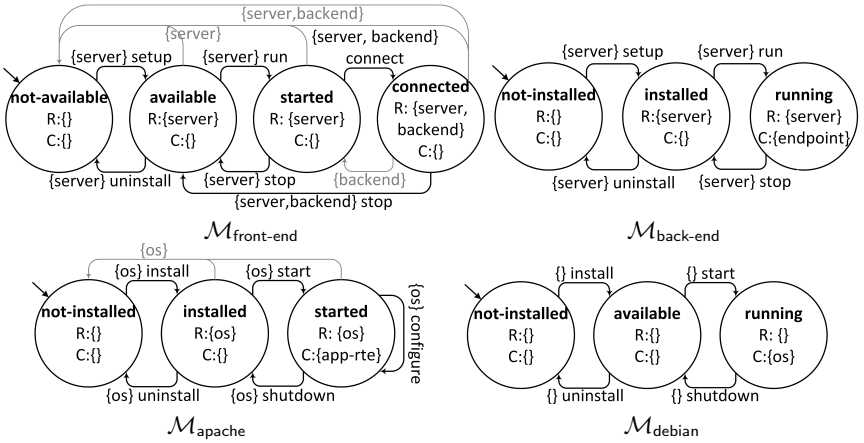


Fig. 2. Examples of management protocols.

[2] The constraints to ensure a deterministic semantics of fault-aware management protocols can be trivially formalised by extending those presented in [4]. In the following we consider management protocols that satisfy such requirements.

Example 1. Fig. 2 shows the management protocols of the nodes composing our motivating scenario (where thicker arrows represent τ, and lighter arrows represent φ).

Consider for instance the management protocol $\mathcal{M}_{\mathsf{apache}}$, which describes the behaviour of the apache node. In its initial state (not-installed) apache does not require nor provide anything. In the installed and started states it instead assumes the os requirement to (continue to) be satisfied. If the os requirement is faulted, then apache returns to its initial state (thus requiring to be installed and started again). The started state is the only one where apache concretely provides its app-rte capability. Finally, note that all apache's operations can be performed only if the os requirement is satisfied.

Consider now the management protocol $\mathcal{M}_{\mathsf{back\text{-}end}}$, which describes the behaviour of the back-end node. When back-end is installed or running, it assumes the capability satisfying its server requirement to (continue to) be provided. What happens if such capability stops being provided? □

The management protocol of a node may leave unspecified how the component will behave in case some requirements stop being fulfilled in some states. To explicitly model that, management protocols can be completed by adding transitions for all unhandled faults, all leading to a "sink" state $s_\lightning$ (that requires and provides nothing)[3].

Definition 2. *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, where* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *is its fault-aware management protocol. The management protocol* $\mathcal{M}_N$ *can be* completed *by replacing* S_N *and* φ_N *with:*

- $S'_N = S_N \cup \{s_\lightning\}$, *with* $s_\lightning \notin S_N$ *and* $\rho(s_\lightning) = \chi(s_\lightning) = \varnothing$, *and*
- $\varphi'_N = \varphi_N \cup \{\langle s, F, s_\lightning \rangle \mid s \in S_N \wedge \varnothing \neq F \subseteq \rho(s) \wedge \nexists \langle s, F, s' \rangle \in \varphi_N\}$.

In the following we will assume fault-aware management protocols to be automatically completed as defined above.

Example 2. The completion of the management protocol $\mathcal{M}_{\mathsf{back\text{-}end}}$ (Fig. 2) is shown in Fig. 3: We add a "sink state" back-end$_\lightning$, and two transitions reacting to the unsatisfaction of the server requirement when back-end is in its installed or running states.

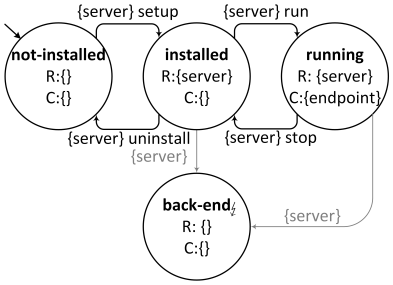


Fig. 3. Example of completed management protocol.

The extension of the other management protocols in Fig. 2 is even simpler: Since they handle all potential faults, their extension only consists in adding a sink state to each of them (i.e., front-end$_\lightning$ is added to the front-end's states, while apache$_\lightning$ and debian$_\lightning$ are added to those of apache and debian, respectively). □

[3] It is easy to prove that the proposed completion preserves the determinism of a management protocol.

3.2 Composition of fault-aware management protocols

Let $A = \langle T, b \rangle$ be a generic composite application, where T is the finite set of nodes (application components) in the application topology[4], and where the connection among nodes is described by a (total) *binding* function

$$b : \bigcup_{N \in T} R_N \to \bigcup_{N \in T} C_N$$

associating each node's requirement with the capability satisfying it.

Since A defines a composition of the nodes in T that coordinate through the binding b among requirements and capabilities, we model the behaviour of A by simply composing the management protocols of the nodes in T.

First, we generalise the notion of global state of A [4] by introducing pending faults. To simplify notation, we shall denote with $\rho(G)$ the set of requirements that are assumed to hold by the nodes in T when A is in G, with $\chi(G)$ the set of capabilities that are provided by such nodes in G, and with $b(R)$ the set of capabilities bound to the requirements in R. Formally:

- $\rho(G) = \bigcup_{N \in T} \{\rho_N(s) \mid s \in G \wedge s \in S_N\}$,
- $\chi(G) = \bigcup_{N \in T} \{\chi_N(s) \mid s \in G \wedge s \in S_N\}$, and
- $b(R) = \bigcup_{r \in R} \{b(r)\}$.

We define the global state of an application A as a set G containing the current state of each of its nodes. We also define a function P to denote the set of pending faults in G, which are the requirements that are assumed in G while the corresponding capabilities are not provided.

Definition 3. *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. A* global state *$G$ of A is a set of states such that:*

$$G \subseteq \bigcup_{N \in T} S_N \ \wedge \ \forall N \in T \colon \exists! s \in G \cap S_N .$$

The set $P(G)$ of pending faults *in G is defined as follows:*

$$P(G) = \{r \in \rho(G) \mid b(r) \notin \chi(G)\}.$$

We denote by $\overline{G}$ the initial global state *of A, where each node of T is in its initial state (viz., $\overline{G} = \bigcup_{N \in T} \{\overline{s}_N\}$).*

The management behaviour of a composite application A is defined by a labelled transition system over its global states, which consists of two simple inference rules, *(op)* for operation execution and *(fault)* for fault propagation.

Definition 4. *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. The* fault-aware management behaviour *of A is modelled by a labelled transition system whose configurations*

[4] For simplicity, and without loss of generality, we assume that, given two nodes in a topology, the names of states, requirements, capabilities, and operations are disjoint.

are the global states of A, and whose transition relation is defined by the following inference rules:

$$\frac{s \in G \quad \langle s, H, o, s' \rangle \in \tau_N \quad P(G) = \varnothing \quad b(H) \subseteq \chi(G)}{G \xrightarrow{o} (G - \{s\}) \cup \{s'\}} \textit{ (op)}$$

$$\frac{s \in G \quad \langle s, F, s' \rangle \in \varphi_N \quad F \subseteq P(G)}{G \xrightarrow{\perp} (G - \{s\}) \cup \{s'\}} \textit{ (fault)}$$

The *(op)* rule defines how the global state of A is updated when a node performs a transition $\langle s, H, o, s' \rangle$. Such transition can be performed when there are no pending faults (viz., $P(G) = \varnothing$), and the requirements needed to perform the transition are satisfied in G (viz., $b(H) \subseteq \chi(G)$). As a result, the global state G is updated with the new state of N (viz., $G' = (G - \{s\}) \cup \{s'\}$), potentially triggering faults to be handled (if $P(G') \neq \varnothing$).

The *(fault)* rule instead models fault propagation. It defines how the global state G of an application A is updated when executing a fault handling transition $\langle s, F, s' \rangle$ of a node N. Such transition can be executed if the faults it handles are pending in G (viz., $F \subseteq P(G)$), and its effects on the whole application A are the following: The state of N is updated (viz., $G' = (G - \{s\}) \cup \{s'\}$), novel faults may be triggered, while the faults in F are not pending any more.

3.3 Analysing an application's fault-aware management behaviour

The management behaviour defined in Def. 4 permits analysing and automating the management of a composite application. For instance, we can easily define which sequences (or, more in general, which workflows) of management operations can be considered *valid* in a global state of an application.

Definition 5. *Let $A = \langle T, b \rangle$ be a composite application. The sequence $o_1 o_2 ... o_n$ of management operations in A is* valid *in a global state G_0 of A iff*

$$\exists G_1, G_2, ... G_n : G_0 \overset{o_1}{\longmapsto} G_1 \overset{o_2}{\longmapsto} G_2 \overset{o_3}{\longmapsto} \ldots \overset{o_n}{\longmapsto} G_n$$

where

$$\frac{G \xrightarrow{o} G'}{G \overset{o}{\mapsto} G'} \qquad \frac{G \overset{o}{\mapsto} G' \quad G' \xrightarrow{\perp} G''}{G \overset{o}{\mapsto} G''}$$

A workflow W orchestrating the management operations in A is valid *in G_0 iff all its sequential traces are valid in G_0.*

Example 3. Consider the workflow in Fig. 4.(a), which permits restarting the back-end and front-end of our motivating application (Figs. 1 and 2). Suppose also that the application is in the following global state: debian is running, apache is started, back-end and front-end are running. It is easy to check that the workflow is valid in the considered global state since both its sequential traces are valid in such global state.

Consider, for instance, the sequential trace performing back-end's stop before front-end's stop. Fig. 4.(b) shows the validity of such a sequential trace by illustrating the evolution of the application's global state. □

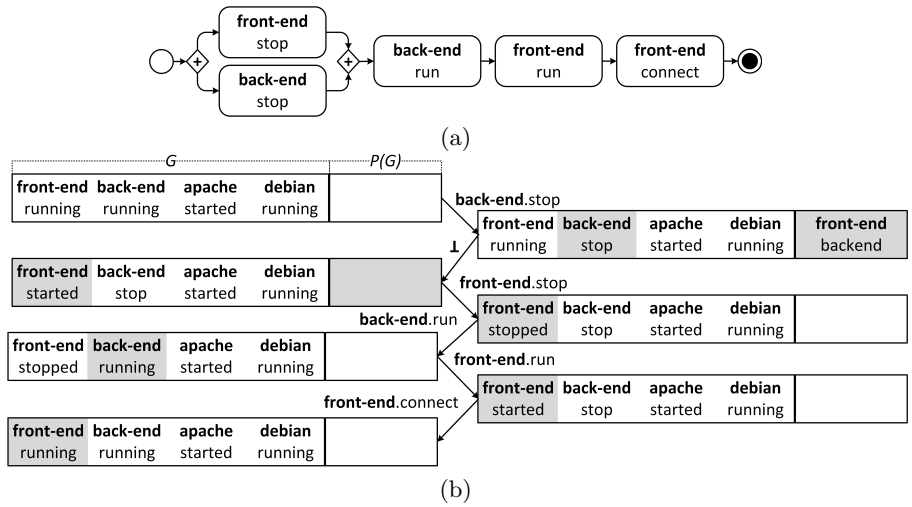


Fig. 4. Examples of (a) valid workflow and (b) valid sequence of operations.

The modelling introduced in Sects. 3.1 and 3.2 can be exploited for various other purposes besides checking whether a plan is valid. For instance, validity may not be enough, as different sequential traces of a plan may reach different global states. It is thus interesting to characterise deterministic plans.

Definition 6. *Let G be a global state of a composite application A. A valid workflow plan W for A is* deterministic *from G if and only if all its sequential traces reach the same global state G'.*

The way to check whether a given plan is valid or deterministic is obviously a visit of the graph associated with the transition system of an application's management behaviour (Def. 4). It is worth highlighting that, thanks to the constraints on management protocols and to the way they are combined, such a graph is *finite* and thus its visit is guaranteed to terminate.

It is also interesting to compute the *effects* of a valid workflow W on the states of an application's components, as well as on the requirements that are satisfied and the capabilities that are available. Such effects can be directly determined from the global state(s) reached by performing the sequential traces of W.

Moreover, the problem of *finding* whether there is a workflow which starts from a global state G and achieves a specific goal (e.g., bringing some components of an application to specific states, or making some capabilities available) can also be solved with a visit of the graph associated with the transition system of an application's management behaviour.

Finally, our model allows to characterise an interesting property that an application may exhibit. If it is possible to reach the initial global state $\overline{G}$ from any global state G that is reachable from $\overline{G}$ itself, then it is always possible to generate a plan for any reachable goal from any reachable global state. This ensures an application's *recoverability*, meaning that whatever global state G

we reach from the initial global state $\overline{G}$ (by executing whatever operation or performing whatever $\perp$-transition), we can always get back to $\overline{G}$, thus always permitting to reset the application.

4 Modelling the unexpected

4.1 Unexpected behaviour of a component

The analysis described in Sect. 3 assumes that each application component behaves according to its specified management protocol, thus not taking into account components that behave unexpectedly because of mismatches between their modelled and actual behaviour (e.g., because of bugs).

The unexpected behaviour of a component can be modelled by automatically completing its management protocol by adding a "crash" operation $\lightning$ that leads the node to the sink state $s_{\lightning}$.

Definition 7. *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is its fault-aware management protocol. The management behaviour of N can be extended to include* unexpected behaviour *by replacing O_N and τ_N with:*

- $O'_N = O_N \cup \{\lightning\}$, *and*
- $\tau'_N = \tau_N \cup \{\langle s, \rho(s), \lightning, s_{\lightning} \rangle \mid s \in S_N\}$[5].

The $\lightning$ operation, combined with the analyses presented in Sect. 3.3, permits analysing the management behaviour of an application also in presence of misbehaving components: Indeed, the possible unexpected behaviour of a node is modelled by $\lightning$ transitions which lead the nodes to their sink state $s_{\lightning}$, where we (pessimistically) assume that the node is not offering any capability any more. This permits us to analyse the (worst possible) effects of a misbehaving node on the rest of the application by simply observing whether and how the global state of the application changes.

Example 4. Consider the back-end's management protocol (Fig. 2), extended by adding back-end$_{\lightning}$ as illustrated in Example 2. The extension described in Def. 2 simply consists in adding "crash" transitions starting from not-installed, installed, and running, and leading to back-end$_{\lightning}$ (Fig. 5). The management protocols of front-end, apache and debian can be extended analogously.

The above extension permits, for instance, determining the effects of a "crashing" back-end when the whole application is up and running (Fig. 6). By invoking back-end's $\lightning$, the global state is changed by updating the state of back-end, and by filling the set of pending faults with the backend requirement of front-end (since it is assumed and connected to the back-end's endpoint capability, which is no more provided). The pending fault is then consumed by a $\perp$-transition, which updates front-end's state. □

[5] $\lightning$ transitions can be fired only if the requirements in $\rho(s)$ are satisfied so as to ensure the well-formedness [4] of management protocols. Note that this is not a restriction since such requirements are satisfied are satisfied in s (by Defs. 1 and 2).

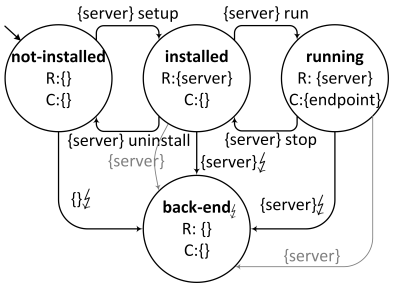


Fig. 5. Example of a management protocol including unexpected behaviour.

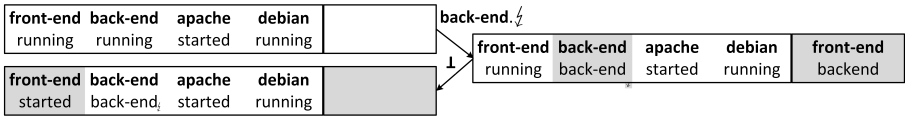


Fig. 6. Example of fault injection and subsequent global state update.

More interestingly, we may wish to recover an application having a component that is behaving unexpectedly. More precisely, from the global state reached after injecting a failure (by invoking the "crash" operation $\lightning$), we may wish to find a "recovery" plan whose execution permits reaching a given recovery goal (e.g., the global state in which the failure was injected). Notice that, such a recovery plan cannot be determined by simply visiting of the graph associated with the labelled transition system modelling the management behaviour of an application, as the faulted node is stuck in its sink state (since no transition outgoes from such state).

4.2 Hard recovery

Recovery plans can be generated automatically, and the underlying idea is quite simple. When a node N is stuck[6] in state $s_{\lightning}$, it can be "hard reset" by the node N' in which it is contained (i.e., by the node in which it is installed or deployed). More precisely, by resetting the container node N', all nodes it contains (among which we have the stuck node N) are forcibly reset to their initial state and can be re-installed and started to return up and running.

Due to space limitations, we hereby show how to recover the global state directly on our motivating example after the fault injection of Example 4.

Example 5. Our objective is to enforce the hard reset of a node N stuck in its sink state $s_{\lightning}$, by restarting the node in which N is contained. This can be naturally modelled with fault-aware management protocols, provided that the topology is extended with an alive capability to explicitly represent the node containment:

[6] In general, hard recovery can be exploited for recovering a desired global state whenever a node is stuck in its sink state.

– Since front-end and back-end are contained in apache, (i) we add an alive requirement to front-end and back-end, (ii) we add an alive capability to apache, and (iii) we connect the alive requirements of front-end and back-end alive requirement with the alive capability of apache.
– Since apache is contained in debian, (i) we add an alive requirement to apache, (ii) we add an alive capability to debian, and (iii) we connect the alive requirement of apache to the alive capability of debian.

The updated topology permits to container nodes to witness whether they are still installed (by providing their alive capability), and to contained nodes to check whether they continue to be installed (by assuming their alive requirement). More precisely, it is

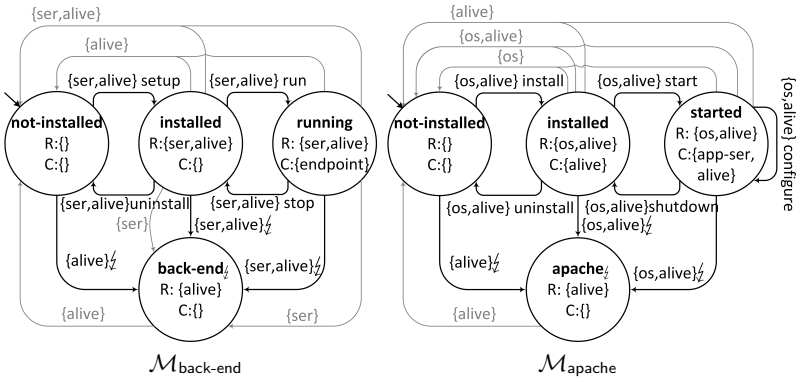


Fig. 7. Example of management protocols with alive requirements and capabilities.

possible to update the management protocols as shown in Fig. 7, which illustrates the updated protocols of back-end and apache[7]. In each state (other than the initial one), apache and back-end assume their alive requirement, i.e. they assume their containers to continue to be installed. apache is also providing its alive capability in such states, to witness to the nodes it contains (i.e., front-end and back-end) that it continue to be there. Additionally, whenever their alive requirement is removed, apache and back-end return to their initial state. This models the fact that, whenever a container is uninstalled, the nodes it contains are uninstalled along with it.

With the above updates we are now able to recover the application from the stuck global state in Fig. 6. Essentially, back-end is stuck in back-end$_{\not z}$, and the only way to get out of it is to remove its alive requirement, which in turn means to shutdown and uninstall apache (to make it stop providing its alive capability). This results in killing also the front-end, which goes back to its initial state. Afterwards, we can re-install and start the apache server, setup and run the back-end and front-end nodes, and connect the front-end (to the back-end).

It can be easily verified that the above listed operations build up a valid workflow permitting the application to be again up and running from the global state reached in Example 4. As we already mentioned, such a plan can simply be determined with a visit

[7] We omit the updated protocols of front-end and back-end due to space limitations, and since their update is analogous to that of back-end and apache.

of the graph associated with the transition system defined by management behaviour of the application. The only requirement is to use a modelling of the applications that is updated as we discussed at the beginning of this example. □

5 Related work

The problem of automating composite application management is one of the major trends in today's IT [18]. Our previous work [4,5], as well as Aelous [10], permit automatically deploying and managing multi-component cloud applications. The underlying idea of both approaches is quite simple: Developers describe the behaviour of their components through finite-state machines, and such descriptions can be composed to model the management behaviour of a composite application. Engage [12] is another approach for processing application descriptions to automatically deploy applications. The approach presented in this paper extends [4,5], and differs from [10] and [12], since it permits explicitly modelling faults and injecting failures in application components, analysing their effects, and reacting to them to restore a desired application state.

The rigorous engineering of fault-tolerant systems is a well-known problem in computer science [6], with many existing approaches targeting the design and analysis of such systems. For instance, [15] proposes a way to design object-oriented systems by starting from fault-free systems, and by subsequently refining such design by handling different types of faults. [22] and [2] instead focus on fault-localisation, thus permitting to redesign a system to avoid the occurrence of such a fault. These approaches differ from ours because they aim at obtaining applications that "never fail", since all potential faults have been identified and properly handled. Our approach is instead more recovery-oriented [7], since we focus on applications where faults possibly occur, and we permit designing applications capable of being recovered.

Similar considerations apply to [13], [16], and [1], which however share with our approach the basic idea of modelling faults in single components and of composing the obtained models according to the dependencies between such components (i.e., according to the application topology).

[11] proposes a decentralised approach to deploy and reconfigure cloud applications in presence of failures. It models a composite applications as a set of interconnected virtual machines, each equipped with a configurator managing its instantiation and destruction. The deployment and reconfiguration of the whole application is then orchestrated by a manager interacting with virtual machine configurators. [11] shares with our approach the objective of providing a decentralised and fault-aware management of a composite application, by specifying the management of each component separately. However, it differs from our approach since it does permits specifying inter-component dependencies, but it is not possible to describe whether they are "horizontal" (i.e., a component requires another to be up and running) or "vertical" dependencies (i.e., a component is installed/deployed on another). Additionally, it focuses on recovering virtual machines that have been terminated because of environmental faults, while we also permit describing how components react to application-specific faults.

[19] proposes an approach to identify failures in a system whose components' behaviour is described by finite state machines. Even though the analyses are quite different, the modelling in [19] is quite similar to ours. It indeed relies on a sort of requirements and capabilities to model the interaction among components, and it permits "implicitly" modelling how components behave in presence of single/multiple faults. Our modelling is a strict generalisation of that in [19], since a component's state can change not only because of requirement unsatisfaction but also because of invoked operations, and since it permits "explicitly" handling faults (i.e., fault handling transitions are distinct from those modelling the normal behaviour of a component). Similar considerations apply to [8], whose modelling is also based on finite state machines with input and output channels (which permit fault communication and propagation by components).

UFIT [14] is a tool for verifying fault-tolerance of systems. It permits modelling systems' behaviour with timed automata, some of whose transitions explicitly represent how the system reacts to the occurrence of faults. Even if it models fault transitions in a way similar to ours, UFIT differs from our approach since it targets standalone systems and does not provide any mechanism to easily compose the automata modelling the behaviour of multiple systems.

Other approaches worth mentioning are [17] and [20]. The way in which our approach models fault-awareness by relying on the interactions between components, as well as the idea of analysing/recovering faults through sequences of atomic transactions (until a desired state is reached), are indeed inspired by [17]. Instead, the idea of relying on fault injection to determine the effects of unpredictable faults is inspired by [20].

In summary, to the best of our knowledge, the approach we propose in this paper is the first that permits automatically orchestrating the management of composite applications under the assumption that faults possibly occur during such management, thus requiring to explicitly model how an application reacts to their occurrence. It does so by following the common idea of modelling each component separately, and of deriving the management behaviour of a composite application by properly combining the behaviour of its components.

6 Conclusions

Management protocols [4] are a modular and reusable way to model the management behaviour of application components, and to automate the management of a complex applications composed by multiple components.

In this paper we have extended management protocols by taking into account the possibility of faults suddenly occurring, as well as of misbehaving components. More precisely, we have shown how to include faults in management protocols, and how to model components' unexpected behaviour. We have also shown how to derive the fault-aware management of a composite application by simply composing the protocols of its components. Finally, we have discussed how the proposed modelling permits automating various analyses (e.g., determining whether a workflow orchestrating the management of an application is

valid, which are its effects, whether it generates faults, or recovering an application that is stuck because of a faulted/misbehaving node).

The presented approach can be exploited for developing engines capable of automatically orchestrating the management of composite application in a fault-resilient manner. Indeed, given a desired application configuration, an orchestrator can automatically execute the sequence of operations needed to reach such configuration, and it can maintain such configuration even if faults or unexpected behaviours suddenly occur.

Please note that, even if some of the analyses we presented in Sects. 3 and 4 have exponential time complexity in the worst case, they still constitute a significant improvement with respect to the state-of-the-art, as currently the management of the components of a complex application is coordinated manually (e.g., by developing ad-hoc scripts), and it is hardly reusable since it is tightly coupled to such application.

It is important to observe that our approach can be easily adapted to cope with applications whose topologies are dynamic. Indeed, to deal with applications whose components may dynamically (dis)appear, we only need to add such components to the application topology, and to update the binding function relating requirements and capabilities. A formalisation of what above is in the scope of our immediate future work.

Another (obvious) extension for future work is to validate the approach we presented in this paper on concrete case studies. In this perspective, we plan to provide tools permitting to model and analyse fault-aware management protocols. More precisely, we plan to (i) properly extend Barrel [4], an open source tool exploiting management protocols for describing and analysing the management of composite TOSCA applications, and (ii) to develop a prototype automatically generating concrete workflows orchestrating the fault-aware management of TOSCA applications.

We also plan to extend the analyses that can be performed on fault-aware management protocols. For instance, we plan to devise techniques permitting to improve our analyses by determining fragments of the topology that can be managed independently from the rest of the topology. This would permit a smarter and more efficient reasoning, as the search space could be reduced by focusing only on the interested fragment(s).

Acknowledgements. This work has been partly supported by the project *Through the fog* (PRA_2016_64) funded by the University of Pisa.

References

1. Alhosban, A., Hashmi, K., Malik, Z., Medjahed, B., Benbernou, S.: Bottom-up fault management in service-based systems. ACM Trans. Internet Technol. 15(2), 7:1–7:40 (2015)
2. Betin Can, A., Bultan, T., Lindvall, M., Lux, B., Topp, S.: Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. Automated Software Engineering 14(2), 129–178 (2007)

3. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: Automated Discovery and Maintenance of Enterprise Topology Graphs. In: Proc. of the 6th SOCA. pp. 126–134. IEEE (2013)
4. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Proceedings, Lecture Notes in Computer Science, vol. 9306, pp. 19–33. Springer International Publishing (2015)
5. Brogi, A., Canciani, A., Soldani, J., Wang, P.: Modelling the behaviour of management operations in cloud-based applications. In: Moldt, D. (ed.) Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE'15. CEUR Workshop Proceedings, vol. 1372, pp. 191–205. CEUR-WS.org (2015)
6. Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E.: Rigorous Development of Complex Fault-Tolerant Systems. LNCS, Springer (2007)
7. Candea, G., Brown, A.B., Fox, A., Patterson, D.: Recovery-oriented computing: Building multitier dependability. Computer 37(11), 60–67 (2004)
8. Chen, L., Jiao, J., Fan, J.: Fault propagation formal modeling based on stateflow. In: Proc. of the 1st ICRSE. pp. 1–7. IEEE (2015)
9. Cook, R.I.: How complex systems fail. University of Chicago (1998)
10. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a Component Model for the Cloud. Information and Computation pp. 100–121 (2014)
11. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. Journal of Systems and Software (2015), *[In press]*
12. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A deployment management system. In: Proc. of the 33rd PLDI. pp. 263–274. ACM (2012)
13. Grunske, L., Kaiser, B., Papadopoulos, Y.: Model-driven safety evaluation with state-event-based component failure annotations. In: Proc. of the 8th CBSE. pp. 33–48. Springer (2005)
14. Hajisheykhi, R., Ebnenasir, A., Kulkarni, S.S.: UFIT: A tool for modeling faults in UPPAAL timed automata. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) Proc. of the 7th NFM. pp. 429–435. Springer (2015)
15. Johnsen, E., Owe, O., Munthe-Kaas, E., Vain, J.: Incremental fault-tolerant design in an object-oriented setting. In: Proc. of 2nd APAQS. pp. 223–230 (2001)
16. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new component concept for fault trees. In: Proc. of the 8th SCS. pp. 37–46. Australian Comp. Soc., Inc. (2003)
17. de Lemos, R., Fiadeiro, J.L.: An architectural support for self-adaptive software for treating faults. In: Proc. of the 1st WOSS. pp. 39–42. ACM (2002)
18. Leymann, F.: Cloud computing. it - Information Technology 53(4), 163–164 (2011)
19. Liggesmeyer, P., Rothfelder, M.: Improving system reliability with automatic fault tree generation. In: Proc. of the 28th FTCS. pp. 90–99. IEEE (1998)
20. Nagatou, N., Watanabe, T.: A model-checking based approach to robustness analysis of procedures under human-made faults. In: Ouyang, C., Jung, J.Y. (eds.) Proc. of the 2nd AP-BPM. pp. 117–131. Springer (2014)
21. OASIS: Topology and Orchestration Specification for Cloud Applications. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf` (2013)
22. Qiang, W., Yan, L., Bliudze, S., Xiaoguang, M.: Automatic fault localization for BIP. In: Li, X., Liu, Z., Yi, W. (eds.) Proc. of the 1st SETTA. pp. 277–283. Springer (2015)