Fault Tolerant Software Intensive System using Distributed Dynamic Tree Logic

CHANDRASEKARAN SUBRAMANIAM¹, RAJALAKSHMI BHAVANISHANKAR²
Department of Computer Science and Engineering
Rajalakshmi Engineering College, Anna University¹
Dr.M.G.R. Education and Research Institute University²
Chennai, Tamilnadu
INDIA
chandrasekaran_s@msn.com, sathwi@yahoo.co.in

Abstract: - The objective of the work is to propose a fault tolerant model for a distributed software intensive system (SIS) employing virtual components with self adaptation techniques. The issues and challenges to achieve the best performance of any software intensive system are to satisfy the unpredictable demands and concurrent requests from various subsystems and accept the heterogeneous data formats. Distributed Dynamic Tree (DDTL) Logic is proposed for the information processing, communication and management of such requests in any software intensive system. The reliability of such a system can be improved by not activating the incorrect services and activating the correct services in time through the integration of communication networks and context aware computing technologies in a highly dynamic environment. The requests can be met not only by replicating some critical subnets of the heterogeneous network but also employing self monitoring mechanisms using Bayesian decision techniques. The various protocols and the routing policies are selected based on the severity and time of the requesting services and the availability of responding services.

Key-Words: - Software intensive, Distributed dynamic, Context awareness, Heterogeneous network, Virtualization.

1 Introduction
A distributed software-intensive system has to interact with other software subsystems, devices and sensors for successful completion of the submitted tasks through concurrent requests. These systems are playing an increasingly dominant role in daily activities like web content mining and service portals. As the complexity of the customer demand is increasing, the needs for software-intensive systems are more that are distributed, heterogeneous, decentralized and interdependent. The systems are operating in dynamic and often unpredictable environment. The formal methods can be used for validation and verification of self-adaptive software intensive systems to ensure its functionality and to understand its behaviour since the specifications are not only used in developing self adaptive software, but also kept for conflict resolution and change management at the operating phase. Goal-oriented requirements engineering appears to be a promising approach for addressing the design challenges in their interfaces and architecture for knowledge sharing inside the system. Generally, most of the proposed solutions do not address the relationships between their own or self properties including priority, conflict and the execution order of their actions at run-time. It is clear that coordinating and orchestrating these properties and their derived goals at different levels of granularity is one of the significant challenges in self-adaptive software intensive systems. In addition to the necessity of deciding dynamically on-line and partially optimal solutions for multi-objective decision-making problems based on their uncertainty and incompleteness of events or information from the system’s self and context. There is also a challenge in correlating local and global decision-making mechanisms and addressing the scalability and fault-proneness of the decision-making mechanism using centralized or decentralized models. Fulfilling global requirements and self properties, for each property and across different properties, is not a straight-forward task [1]. To manage uncertainty in computing systems and their environments, there is a need to introduce feedback loops to control the uncertainty. An example for a self-adaptive system following the MIAC scheme applied to software is the robust feedback loop used in self-optimization that is becoming prevalent in performance-tuning and resource-provisioning scenarios [2]. Any software-intensive product and service providers, especially from the secondary industry must deal with a two-way-alignment between the business problem and
the corresponding software solution by standardization of components and semantic component interfaces to allow semantic run-time evaluations. Robyn Lutz provided a roadmap to formal methods for developers of safety-critical systems by further integration of informal and formal methods [3]. In a distributed software intensive system, a feature interaction occurs when a combination of several features that work well in isolation causes contradictions or unexpected system behavior in the context of complex software intensive electronic control systems [4]. There is a need for a methodology for specifying and verifying software intensive systems. Developing such an approach poses a grand challenge when several simultaneous activities, occasionally synchronizing and assembling them at specific milestones to ensure that the final product successfully comes out [5]. The earlier works extended by adding an effectiveness factor that reflects the investment in software engineering tools, processes and code expansion that makes the work of one programmer more effective [6]. The current state-of-practice using fault tree analysis (FTA) and failure mode effect analysis (FMEA) are not sufficient to detect faults in the complex and interconnected modern systems. Many models were proposed based on formal verification for capturing and checking requirements. The main objective is to apply formal methods and tools for engineering adaptive software intensive systems to ensure the required levels of reliability and availability with expected level of correctness and security. The composition of various asserted services and their soft real time behaviour makes a software system as a fault tolerant one.

The paper is organized as follows: Section 2 declares the essential features of a fault tolerant software intensive system in terms of processing, communication and management of critical information and the sub section 2.1 discusses the need for a multi software activation model. Section 3 discusses the proposed dynamic distributed logic and its verification through sequent calculus. Section 4 brings out the adaptability of the software intensive system with deontic logic for decision making and Section 5 addresses the identity management issues with the timing diagram of the request instances to complete task in the presence of faults to minimize the timing complexity to take the correct decision in the case of faulty situations. Section 6 concludes the work with its limitations and future work on Fault tolerant SIS.

2 Fault Tolerant Software Intensive System

The software intensive systems have to adapt according to the unforeseen changes in the requirements, technology, environment and operating conditions. Also the quality and trust with which the systems can interact with other subsystems should be declared to maintain the required quality including those that derives from operational and normative behavior of other systems. The trust and reputation has to be preserved up to an accepted level while guaranteeing reliability. The various components with their interoperability are to be collaborated even with only a partial knowledge of their operational environment. The correct and fast response of such intensive systems is needed during emergency conditions and these systems are to be built where a useful coordinated behavior emerges as a result of interactions of independent parts. A software intensive system can be modeled as a complex process where each software component is performing a relatively low complexity process. Many components may be triggered to perform parallel processing on restricted communication channels. The output of one process can react with the inputs of many components in parallel and the parallel composition takes certain delay. The input may be replicated many times to interact with other components using secured interfaces to form a nesting of processes. The communication between multitudes of component processes can be achieved by sending dedicated objects through the channel. The channel may be a physical or a virtual one to establish a virtual network of replicated components or a physical network with installed components in the respective subsystems. Parallel and sequential composition, replication, virtualization and binding processes are to be considered for the given complex SIS. The features that are needed are completeness over time with security over individual processes so as to complete within the stipulated time.

The fault tolerant SIS may be modeled and designed with the focus to avoid or tolerate the faults mainly through due to:

- Dynamic decision making
- Adaptive components
- Identity management for services

The synchronization in data transfer helps to estimate not only the availability, but also the performance of the system. The specification fault occurs when the parallel composition of two or
more processes is called and if the specified number of given components are not replicated correctly. The interfacing fault occurs when input and output actions are performed on a common noisy channel. The synchronization fault occurs during rollback mechanisms if the input or the rollback does not synchronize the output of that rollback stage. Due to all these faults and their consequences the distributed system may fail to terminate or execute. Reliability is the ability of a software component or system to consistently perform according to its specifications in the target operational environment. Hence a software intensive system can be designed in a framework where three different models can be used. They are information processing model, information communication model and information management model.

<table>
<thead>
<tr>
<th>SIS feature</th>
<th>IP model</th>
<th>IC model</th>
<th>IM model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification</td>
<td>Process</td>
<td>Channel</td>
<td>Activation</td>
</tr>
<tr>
<td>Connection</td>
<td>Binding</td>
<td>Fission</td>
<td>Regulation</td>
</tr>
<tr>
<td>Function</td>
<td>Action</td>
<td>Interaction</td>
<td>Computation</td>
</tr>
<tr>
<td>Changeover</td>
<td>Transform</td>
<td>Preservation</td>
<td>Replication</td>
</tr>
<tr>
<td>Conversion</td>
<td>Switching</td>
<td>Differentiation</td>
<td>Coding</td>
</tr>
<tr>
<td>Collaboration</td>
<td>Composition</td>
<td>Reassembly</td>
<td>Auditing</td>
</tr>
<tr>
<td>Specification</td>
<td>Hiding</td>
<td>Consuming</td>
<td>Inhibition</td>
</tr>
<tr>
<td>Outputting</td>
<td>Reporting</td>
<td>Pushing</td>
<td>Decision</td>
</tr>
</tbody>
</table>

Table 1 SIS feature

IP: Information processing
IC: Information communication
IM: Information management

2.1 Multi-Software Activation in SIS

The completeness and safety properties of software intensive system are based on the channel activity which corresponds to number of possible combination of inputs and outputs triggered by deployed software components on the given channel. In this scenario an execution environment is considered to be a platform that supports a multi-software activation with number of software components, services, packages and objects. In this activation model, two types of communication are possible. The communications that take place between services or packages are termed as inter software component communication through dedicated interfaces. During the communication between software components naturally a pathway is created for movement of objects. A functional pathway will ensure a smooth flow of information between components, whereas a quality pathway will ensure timed and secured activations. This results in a synchronization reaction for asynchronization actions.

Consider the scenario1: “If the application is failed or the network is unavailable then activate the redundant application within the physical machine through application migration from one virtual machine to another virtual machine or establish the virtual private network”.

Its equivalent First Order Logic (FOL) is given by,

$$\exists t \, \text{Failure}(A, t) \lor \text{Failure}(N, t) \Rightarrow M_A(V_1, V_2) \lor \text{Connect}(VPN)$$

where, A represents Application, N represents Network, M_A represents Migrate application, VPN represents Virtual Private Network, V_1 & V_2 represent virtual machines and t represents time. The above representation fails to mention that the “redundant” application should be migrated.

Consider a scenario2: “If more number of components are to be collaborated for the decision making then establish trust between the components and based upon the context of the request, deploy a more secured application in the scenario”.

Its FOL is given as:

$$\exists c \, D_M \Rightarrow \text{Collaborate}(c_1, c_2, \ldots c_n) \Rightarrow \text{Trust}(c_1, c_2, \ldots c_n) \land \text{Deploy}(S_A)$$

where c represents components in the system, D_M represents decision making and S_A represents secured application. The above representation fails to specify that the secured application has to be deployed based on the “context of the request”.

Consider a situation: “If any one service components fails in a federated environment, the system must self organize dynamically to ensure a high reliability based on situation and time awareness. The specification of the component and the available interfaces should be adapted to ensure integration of all communication networks to complete the tasks”.

$$\exists c \, \text{Failure}(c, F_E) \Rightarrow \text{Organize}(S, d) \land \text{Complete task}(S) ;$$

$$\text{Organize}(S, d) \Rightarrow \exists t \, \text{Include}(s_A, S) \land \text{Include}(t_A, t);$$

$$\text{Complete task}(S) \Rightarrow \text{Adaptive}(c, S) \land \text{Adaptive}(I, S) \Rightarrow \text{Integration}(C_N);$$

where F_E represents federation environment, S represents system, s_A represents security awareness, t_A represents time awareness, t is the time at that instance, C_N represents communication networks. To achieve fault tolerant system behavior, for example considering the statement, “The virtual applications
(V_A) should be immediately migrated to another physical server (P_S) and then the file transfer should be carried out in the correct sequence as per the order of the business mail. The emergency requests (E_R) from now onwards should be preserved until the confidentiality policy (C_P) changes when the proxy server is at location1 or location2.”

\[ \text{Migrate}(V_A, P_S) \land \text{File Transfer (Sequence)} \rightarrow \neg \text{Change (C_P)} \rightarrow \text{Process (E_R)} \land \text{Proxy(_loc1)} \lor \text{Proxy(_loc2)} \]

The FOL does not allow us to represent the action “redundant” in the first statement, “context of request” in the second statement and the action “next” in the third statement. FOL allows the representation of the statements in a very generic manner; it supports only a limited number of specific conditions without focusing the urgent operations and its criticality. The software intensive systems need an application or situation specific logic to address the need for high reliability. Distributed dynamic tree logic is proposed to solve these issues in the next section.

### 3 Distributed Dynamic Tree Logic

A Distributed Dynamic Tree Logic is proposed along with the existing operators in LTL and CTL in a ‘do and don’t’ style.

\[
\varphi = \mathcal{T}[p] D_o \varphi | D_N \varphi | (\mathcal{T}_{1} \varphi) \mathcal{U} \varphi \mathcal{A} | N_o \varphi | X \varphi
\]

where

\[
D_o \rightarrow \text{Do}, \ D_N \rightarrow \text{Don’t do}, \ N_n \rightarrow \text{Now onwards} \ U \rightarrow \text{Until}, \ I \rightarrow \text{Immedately} \ X \rightarrow \text{Next} \ A \rightarrow \text{Always} \ \varphi \rightarrow \text{Set of propositional atoms}.
\]

In case of a software intensive system like emergency management system there are some do’s and don’ts that have to be followed at that instant of time when some incident occurs. The above mentioned CTL and LTL cannot be used to represent the urgent nature of the events which may be categorized as events that should be done or should not be done like do’s and don’ts. The proposed DDTL is used to represent the liveliness and safety properties of various actions that are to be taken. Some global system or local variables are to be kept invariant when called for such component services are called for the safety property and especially the local variables are to be checked for their termination or correct execution for liveliness property of the system. A linear temporal logic can be used to specify one path. The linear temporal logic LTL considers temporal operators that refer not only to the future but also to the past where the future can be represented with unary operators X and U. Hence new operators may be incorporated such that the intensive system can perform some actions based on the decision taken among these conditional logics in emergency.

<table>
<thead>
<tr>
<th>Do’s</th>
<th>Don’ts</th>
<th>Symbolic Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do the action</td>
<td>Don’t do the action</td>
<td>D_o.a</td>
</tr>
<tr>
<td>Do action immediately</td>
<td>Don’t do action immediately</td>
<td>D_o.I.a</td>
</tr>
<tr>
<td>Do action ( a_i ) until an event ( e_j ), ( i \neq j )</td>
<td>Don’t do any action ( a_i ) until event ( e_j ), ( i \neq j )</td>
<td>D_o.a_i \ U.e_j</td>
</tr>
<tr>
<td>Always do</td>
<td>Never do</td>
<td>A.D_o</td>
</tr>
<tr>
<td>Now onwards do</td>
<td>Now onwards don’t do</td>
<td>No.D.o</td>
</tr>
<tr>
<td>Do in sequence</td>
<td>Don’t do in sequence</td>
<td>D_o.i ; D_o.j ; D_o.k ( i &lt; j &lt; k )</td>
</tr>
<tr>
<td>Do this or that</td>
<td>Don’t do this or that</td>
<td>D_o.i \lor D_o.j ( i \neq j )</td>
</tr>
<tr>
<td>Do it next</td>
<td>Don’t do it next</td>
<td>D_o.X</td>
</tr>
</tbody>
</table>

The relationships between the do’s and don’ts operations can be summarized by the following equivalence or tautologies to derive checking conditions for a software intensive system.

\[
D_o.\varphi \equiv \sim D_N.\varphi
\]

\[
D_o.I.\varphi \equiv \sim D_N.I.\varphi
\]

\[
D_o.U.\varphi \equiv \sim A.D_O.\varphi
\]

\[
N_o.D_o.\varphi \equiv \sim A.D_O.\varphi
\]

\[
N.o.D_o(t,e).\varphi \equiv D_N.U(t,e).\varphi
\]

\[
\sim (D_o.\varphi \lor D_o.\varphi_2) \equiv D_N.\varphi_2 \land D_N.\varphi
\]

\[
D_o.\varphi_1 ; D_o.\varphi_2 \equiv D_O.\varphi_1 XD_o.\varphi_2
\]

\[
D_o.I.\varphi_1 \equiv D_N.I(\varphi_2 , \varphi_3 , \ldots \varphi_n)
\]

\[
D_o.\varphi_1 \lor D_N.\varphi_2 \equiv D_N.\varphi_2 \lor D_O.\varphi_1
\]

\[
(D_o.\varphi_1 \lor D_o.\varphi_2) \land D_N.\varphi_3 \equiv (D_O.\varphi_1 ; D_N.\varphi_2 ; D_N.\varphi_3)
\]

\[
(D_o.\varphi_1 \lor D_o.\varphi_2) \land D_N.\varphi_3 \equiv (D_O.\varphi_1 ; D_N.\varphi_2 ; D_N.\varphi_3)
\]
The application of the DDTL can be explained in the following scenario: “if the software intensive system faces distributed security faults where the base network may be in emergency (\(BN_E\)) then an application is to be enabled (\(A_e\)) then activating the virtual private network (\(A_{VPN}\)). When a virtual private network is activated service fault is detected (\(D_{SF}\)). The replication (\(R\)) is performed in virtual network and the service fault detection occurs to make the system in safe state (\(S\)).“ The above situation can be represented and verified using sequent calculus and the DDTL equivalences.

\[
\begin{align*}
BN_E, E_a \vdash A_{VPN} & \quad A_{VPN} \vdash D_{SF} \quad \text{Cut Rule} \\
BN_E, E_a \vdash D_{SF} & \quad R \land \\
BN_E, E_a \vdash D_{SF} \land R & \quad L \Rightarrow \\
BN_E \Rightarrow D_{SF}, E_a \vdash R & \quad R \vdash S \quad \text{Cut Rule} \\
BN_E \Rightarrow D_{SF}, E_a \vdash S \\
\end{align*}
\]

If any service component fails \(F(S)\) in a federation environment (\(F_E\)) then the system must self organize dynamically (\(D_{O}\)). Reliability of the system depends on the time and situation awareness of the self organizing components.

\[
\begin{align*}
F(S), F_E \vdash D_{O} & \quad L \exists \\
\exists S \, F(S), F_E \vdash D_{O} & \quad R \land \\
\exists S \, F(S), F_E \vdash D_{O} \land T_A & \quad L \Rightarrow \\
\exists S \, F(S) \Rightarrow D_{O}, F_E \vdash T_A & \quad RW \\
\exists S \, F(S) \Rightarrow D_{O}, F_E \vdash T_A, S_A & \quad T_A, S_A \vdash R \quad \text{Cut Rule} \\
\exists S \, F(S) \Rightarrow D_{O}, F_E \vdash R \\
\end{align*}
\]

4 Adaptive Distributed Model

The complexity, uncertainty and adaptability issues of a distributed SIS (DSIS) are to be identified to design a fault tolerant system. The system reliability may be predicted if dynamic fault tolerant features are incorporated at the component level or service level or even the virtual machine level. The application submitted to the distributed software intensive system may be a process intensive or memory intensive or computing intensive or a combination of more than one. The reliability of such a distributed software intensive system depends on the complexity of any of its component or service. Software intensive system will act as a system of collaborating computing processes in a highly dynamic environment and will be integrated with service oriented and pervasive computing techniques. Conventional formal approaches are not well integrated with pragmatic methods and do not scale up to complex software intensive systems. The important design aspects of any DSIS are due to the changes in the operating environment and its quick adaptation towards the heterogeneous components and the expected quality of service with increased trust. The uncertainties present in the completion of tasks submitted to the system depend on the moral or knowledge with which the components had been designed. Since the components or services are heterogeneous in nature, the predictability of any specific property emerging from the system will be acceptable if there is adaptability incorporated within the system. The adaptation policy is based on the correctness in its completion and composition and it will be based on the trust and security levels. The context aware trust values among the component services can be predicted by bringing a mathematical model as given in the earlier work [8]. The trust value of the system in a given context at a given time can be determined by considering the capacity level and the context level of the incoming. The different capacity levels in the software components are in the entity level, process level, task level or system level. That is, the system will trigger its actions based on the mutual trust between collaborating components and services that can be predicted by adding the previous trust level of all the entities in all contexts and the overall non suspicion value at different capacity levels. The trust evaluation is depending upon both relationship and quality attributes based on reputation. The dynamic nature of adaptation leads to another challenge of identifying the collaborating services where a number of physical and virtual services are deployed and participated. Then each of service and its parameters are to be asserted when provision the service or reprovision the service in the correct server.

5 Identity Management Issues

The identity-management problem in internet computing becomes more crucial in finding out correct application in an inter-organizational setup. The assertions are to be made in composing heterogeneous web services with the identity of the requesting and responding servers, identity of the requesting clients, uptime and downtime of the servers. The waiting time of the clients, whether new services have been added, registration of the newly added services, initial state of the server and
transfer of registration information are to be identified and enabled correctly. For example, the collaborating services may be in following status like Server 1 is up and responding Server 2 is down and has to be restored. The Client 1 and Client 2 are in two timing instances where Client 1 is requesting, Client 2 is waiting and Service 1 is new and is registered are as shown in Fig.1. The identity manager will capture the traces of all the instants of components and events and to decide the necessary action to make the system available making it as a fault tolerant one. The sample tracing done by the event manager component for the concurrent and sequential requests that is shown in Fig.1

Any expected or needed set E of property p evaluates to true for a timed transition if and only if there is a sequence of alternating delay transitions from the starting action to all the action transitions till the final safe transition satisfying that property p. The properties may be completion time of individual processes or composition over a number of processes satisfying say, security of the individual processes. It may be explained with the help of the proposed DDTL by not only doing the necessary action without fail but also not doing the unnecessary actions within that session. By applying timed automata with assertion of the property over various servers deployed.

The time and event management modules wait for the soft real time functions to get completed even in the presence of faults. The intermittent faults and the timing faults are the most serious types of faults as far as the software intensive systems are considered. The intermittent faults are transient in nature that can be tolerated through activation of virtual private networks with a series of “Dos” as shown in Fig.2. The synchronization can be achieved by the timing properties for example as,

\[ \text{Do.I if } x = 0 \text{ && } x < 50 \text{ && Do.X, where I is an immediate action and X is a don’t care action.} \]

The service components are composed according to similar “Do next” and “Never Do” logic variables and their sequences with other “Don’t do” sequence as shown in Fig 3. The service faults detection and the appropriate replication activation can be achieved through the time management component to achieve a successful composition of services from various servers deployed.

Fig.1 Intensive Application Tracing

Fig.2 Fault tolerant Completeness over time

Fig.3 Fault tolerant Composition
The security property which is in the form of verifying the trust values and its associated certificates are also checked in the software intensive system. The “Always” and “Do it now onwards” logic variables are considered to take the decision at appropriate times as shown in Fig 4. The integers in the figure represent the timing constraints for the individual actions to get completed. The system quality attributes are verified through the respective test cases. The trust values between component services and the timing of redundant activation in the case of faults are checked and tabulated as shown in Table 3. For example, from the set of property E, it is checked from the Basenet 3 by enabling the API1 and it activates the virtual network 2 and checks its trust value as given by the syntax shown below:

\[
E< \rightarrow \text{Base net (3). EnableAPI1 and Virtualnet2.TrustCheck}
\]

Table 3 Property Test case Results

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>E&lt;&gt; Base net(0).EnableAPI1</td>
<td>Property is satisfied.</td>
</tr>
<tr>
<td>E&lt;&gt; Basenet(4).VPN1Activation</td>
<td>Property is satisfied.</td>
</tr>
<tr>
<td>E&lt;&gt; Virtualnet1.Replication</td>
<td>Property is satisfied.</td>
</tr>
<tr>
<td>E&lt;&gt; Virtualnet2.InterfaceCheck</td>
<td>Property is satisfied.</td>
</tr>
<tr>
<td>E&lt;&gt; Basenet(1).EnableAPI1 and Virtualnet1.ActivateVPN1</td>
<td>Property is satisfied.</td>
</tr>
<tr>
<td>E&lt;&gt; Basenet(3).EnableAPI1 and Virtualnet2.TrustCheck</td>
<td>Property is satisfied.</td>
</tr>
</tbody>
</table>

6 Conclusion

The software intensive system is modeled and designed to tolerate faults that will occur in activating the incorrect things and not doing the correct things in time by following the DDTL. The model is verified using sequent calculus for its completeness and satisfiability properties. The multi-software activation model is subjected to timing and synchronization faults that are taken care of self adaptability and identity management techniques and the features are asserted with the help of Timed Automata. The reliability and trust of the individual component services are assumed to be varying uniformly not stochastically. The network and application virtualization techniques will be considered to enhance the reliability of such a massively parallel software intensive system in future.

References: