Design Framework for Self-Stabilizing Real-Time Systems Based on Real-Time Objects and Prototype Implementation with Analysis[1]

Sushil S. Digewade and Albert M. K. Cheng

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

UH-CS-04-06
November 15, 2004

Keywords: real-time systems, self-stabilization, object-oriented design, software engineering, performance analysis

Abstract

The paper proposes an object-oriented design framework to implement self-stabilizing real-time systems and describes an implementation and analysis of a prototype of such a system. The paper also includes a brief study of application of the object-oriented paradigms such as reflection and inheritance based on the design framework. The analysis includes a determination of overhead and a feasibility study of the prototype. The framework is three-tiered in which the basic or the firs- tier components are the self-stabilizing components of the distributed real-time systems that recover themselves in case of transient faults. The second- tier components of the design framework consist of the RobustRTOs and the MetaObjects. The third-tier components are the Guards of the subsystems.

[1]This project is supported in part by a 2004 grant from the Institute of Space Systems Operations (ISSO).

Design Framework for Self-Stabilizing Real-Time Systems Based on Real-Time Objects and Prototype Implementation with Analysis

Sushil S. Digewade and Albert M. K. Cheng
Real-Time Systems Laboratory
Department of Computer Science
University of Houston – University Park
Houston, TX 77204-3010, USA
Email: cheng@cs.uh.edu

Abstract

The paper proposes an object-oriented design framework to implement self-stabilizing real-time systems and describes an implementation and analysis of a prototype of such a system. The paper also includes a brief study of application of the object-oriented paradigms such as reflection and inheritance based on the design framework. The analysis includes a determination of overhead and a feasibility study of the prototype. The framework is three-tiered in which the basic or the firs- tier components are the self-stabilizing components of the distributed real-time systems that recover themselves in case of transient faults. The second-tier components of the design framework consist of the RobustRTOs and the MetaObjects. The third-tier components are the Guards of the subsystems.

1. Introduction

Objects, as of the Object oriented paradigm, are the essential building blocks for a component or module based design. This modular design makes the system extensible, reusable and manageable. To obtain these benefits, object-oriented design could also be applied to real-time systems. The conventional objects cannot represent the complex temporal behavior of real-time systems. Instead of the conventional objects, K.H. Kim [Kim97] proposed the use of Real-time objects (RTO) in the object-oriented design of real-time systems.

The essential elements of a RTO are the Object data store, the Message-triggered methods and the Time-triggered methods [Kim97]. The Object data store contains the data required by the RTO, this data store holds the capability to aggregate other RTOs and could act as a communication channel. The Message-triggered methods are invoked by other RTOs mainly as a service request. The Time-triggered methods are periodic methods, which are self-invoked. Figure 1 describes the RTO. To meet real-time constraints, it is necessary to have design-time guarantees on these methods.

With regards to RTO very little has been done to establish a design framework for fault-tolerant real-time systems. This has been the prime motivation for the work described in the paper. The past efforts of establishing design frameworks FRIENDS [Fabre98], ROAFTS[Kim98] and Chamelon [Kalbarczyk99] laid more stress on fault-tolerance aspect rather than real-time system aspects. The most recent real-time object model [Lim00] is the only design framework so far, which is built taking into consideration the dependable or fault-tolerant real-time systems. This design framework [Lim00] is very general as far as the fault-tolerant mechanism is considered. Our work is based on providing a design framework for more specific self-stabilizing [Cheng92] components.

Self-stabilization is the process of automated recovery in distributed rule-base real-time systems in case of transient faults [Cheng92]. These self-stabilizing components are structurally different from general fault-tolerant systems where recovery from faults is achieved through multiple versions of the same

components. The basic elements of a self-stabilizing real-time system are identified to layout the proposed design framework. A prototype self-stabilizing real-time system has been implemented based on the design framework and the analysis involves determining the processing overhead incurred by the framework based implementation. In section 2 the basic components of the proposed framework are explained to get better insight into the details of the framework. In section 3 the layout of the design framework is discussed. In section 4 the implementation of a prototype is described. Section 5 includes the analysis of the implemented prototype. The conclusions are made in the section 6.

2. Basic Elements of the Proposed Design Framework

All design frameworks have different types of building blocks that result into the overall design framework. The proposed design framework has the following elements:

2.1. Self-Stabilizing RTOs

The Self-Stabilizing RTOs are the basic components of the framework. Any RTO that has methods, which can be recognized as rule-based of form A [Cheng92] can be transformed into a self-stabilizing RTO. The details of this transformation can be found in [Cheng92]. Consider a non-self-stabilizing rule set as below:
Rule 1. If (a = 1) then { c = 2; d = 4;}
Rule 2. If (b = 1) then { c = 3; d = 5;}
where Rule 1 and Rule 2 are mutually exclusive. These two rules form a set, which is of the special form A, could be made self-stabilizing by adding the following rule to the above rule set.
If ((a != 1) or (b != 1) then { c = init_val_of_c; d = init_val_of_d;}
where init_val_of_c and init_val_of_d are the initial values of the variables c and d.
It is vital at the design level to specify which RTOs could be targeted as the prospective components for self-stabilization transformation.

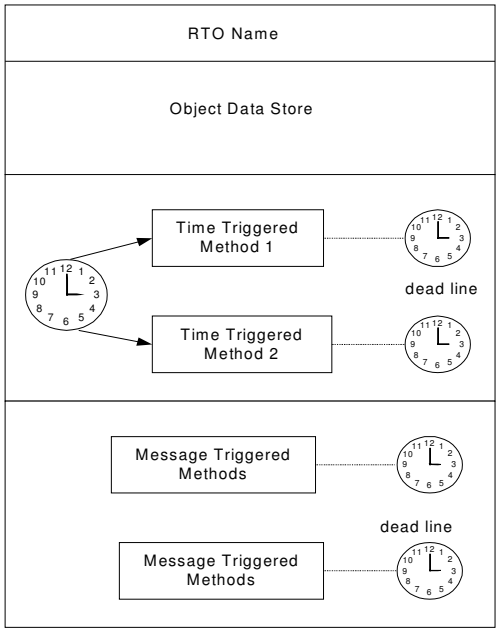

Figure 1. RTO representation

2.2. Metaobjects

Metaobjects are the high-level abstractions of the objects. In this paper, the metaobjects are the high-level abstractions of the RTOs and self-stabilizing RTOs. Metaobjects define the behavior of the underlying RTOs. Along with Reflection, metaobjects are used to define the metaobject protocol (MOP) [Mitchell97]. Reflection is the age-old method of the object-oriented paradigm for getting the information out of objects. Java has built-in reflection methods that can give information about the object at run-time.

MOP can be used to give the application developer better understanding of the underlying relevant properties of the RTOs while maintaining the language implementation. Irrespective of the underlying implementation, the real-time properties of the RTOs could be preserved using the metaobjects. The relevant properties of the underlying RTOs could be obtained using the Reflection on the metaobjects. The worst-case execution time, deadline, periodicity, etc., are some of the real-time properties of the RTOs that could be obtained from the metaobjects.

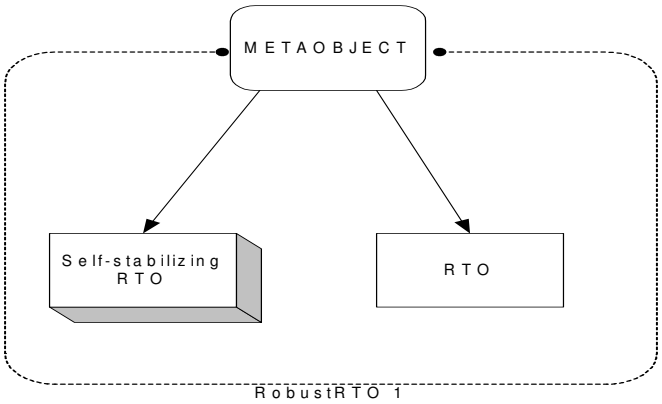

Figure 2. A self-stabilizing RobustRTO

2.3. RobustRTOs

RobustRTO is a fault tolerating RTO [Lim00]. According to Lim and Yang a RobustRTO encapsulates one or more replica RTOs as shown in figure 2 and they proposed it for an N-version fault tolerant mechanism. In case of N-version fault-tolerant systems the components are themselves not fault-tolerant, it is the mechanism or scheme as a whole that is fault-tolerant. Hence, RobustRTO was introduced to separate the fault-tolerance implementation from the application implementation. In case of self-reliable components there is no external fault-tolerance method that is involved unless a spare backup component is used for protecting against the node failure of the primary self-reliable component. The method to implement the spare

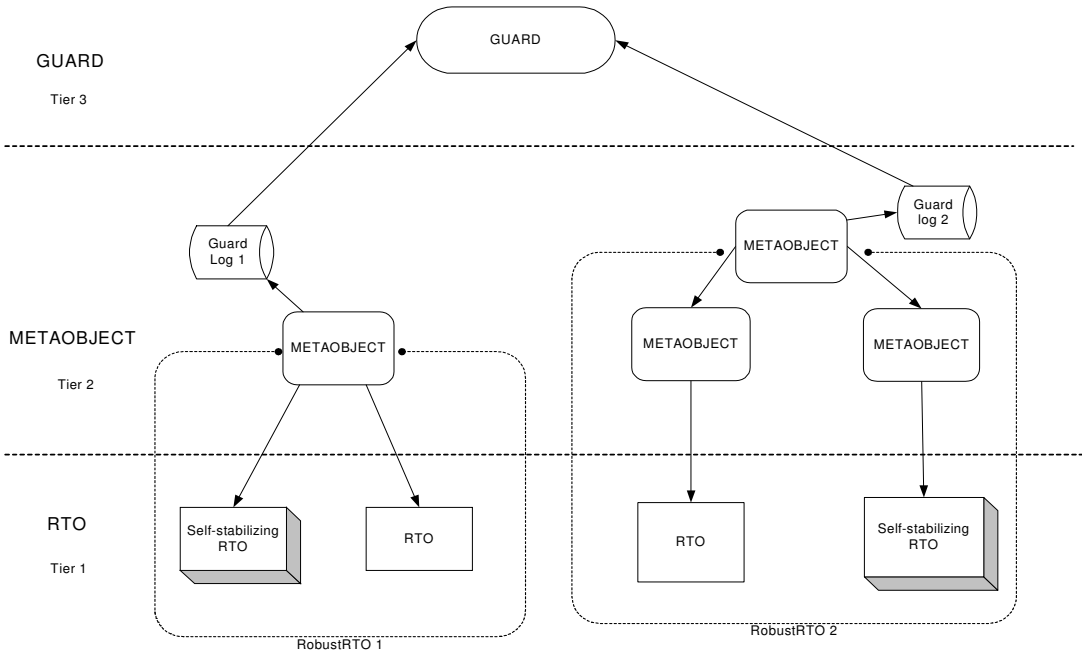

Figure 3. The overall design framework scheme of things

component can be efficiently implemented using the metaobjects since metaobjects are designed to accommodate the run-time changes. Hence, in this paper RobustRTO is just a design level abstraction nevertheless important to modularize the different components.

2.4 Guards

Gurads are similar to the Monitors [Lim00] in the sense that they monitor the reliable real-time subsystems. A subsystem is a logical part of the whole system derived on the basis of its functionality. However, the functions of the Guard are different from the Monitor. A self-stabilizing RTO subsystem is itself reliable as far as the software transient faults are considered. Hence, in the self-stabilizing RTO subsystem, the transient faults are contained within the RobustRTOs. The primary function of the Monitors is to localize the transient faults to a subsystem where as the primary function of the Guard is to monitor each subsystem component for nodal failure.

Each self-reliable RobustRTO component maintains its own guard_log, a log file, which gives information to the Guard about the nodal failure of a self-stabilizing component. In case of the nodal failure the Guard has to take proper action. This action could vary depending upon the implementation of the subsystems; the Guard can shutdown the subsystem or invoke another backup subsystem.

3. Design Specifications of the Components

The overall design framework for the self-stabilizing RTO model is shown in figure 3. The scheme corresponds to the elements described in section 2 and can be replicated to derive a complex real-time system. In other words, the framework is scalable. From design perspective, it is necessary for a design team to come up with a design specification for the development team. Hence it is necessary to specify the design framework for this specification of the components described in the section 2.

3.1. Specification for Self-stabilizing RTOs

The specification for a self-stabilizing RTO is shown in figure 4. The specification denoted in regular

expression includes Data Store, Time Triggered Methods and Message Triggered Methods. The DataStore contains the name of other aggregated RTOs and the data elements denoted by <dataName>. The <dataName> as seen in the figure 4 is composed of the <validTime> notation that specifies the time for which the data is valid.

```
RTO name {
        DataStore:
                <name of aggregated RTOs>* | < dataName>*
                {<dataName> := <name><validTime>}
        TimeTriggeredMethods:
                <TMethodName>*
                {<TMethodName> := <name> <periodicity><selfstabilizingStatus>}
        MessageTriggeredMethods:
                <MMethodName>*
                {<MMethodName> := <name> <deadline><selfstabilizingStatus>}}
```

Figure 4. Specifications for RTO

There can be one or more Time triggered and Message Triggered methods as denoted. The <selfstabilizingStatus> notation denotes whether the method can be targeted for the process of self-stabilization. The <periodicity> denotes periodic time interval and the <deadline> indicates the deadline of the task.

Apart from the above-described elements, the self-stabilizing RTO spec can include the <msg-list> and the <caller> | <callee> [Lim00] that denote the messages sent to this RTO and the caller of this RTO, respectively.

3.2. Specification for Metaobjects

The Specification for the Metaobjects is described in figure 5. The specification framework for the metaobjects is similar to the RTO specification but with the addition of the Member and PropertyMethods notations. The Member notation denotes the name of the underlying RTOs, which the metaobject represents. The PropertyMethods notation defines the methods that give the information regarding the real-time properties such as the deadline, WCET, periodicity, etc.

Properties such as current time, slack time, freshness of data stored, etc., that influence the real-time systems have been identified previously [Mitchell97]. Many scheduling policies require run time information and can be retrieved using reflection through the PropertyMethods of the metaobjects.

```
Metaobject Name {
        Member:
                <name of Underlying RTOs>*
        DataStore:
                <name of aggregated RTOs>* | < dataName>*
                {<dataName> := <name><validTime>}
        TimeTriggeredMethods:
                <TMethodName>*
                {<TMethodName> := <name> <periodicity><selfstabilizingStatus>}
        MessageTriggeredMethods:
                <MMethodName>*
                {<MMethodName> := <name> <deadline><selfstabilizingStatus>}
         PropertyMethods:
                <PMethodName>*
```

```
        {<PMethodName> := <name><description>}
}
```

Figure 5. Specifications for the Metaobject

3.3. Specification for RobustRTO

For the proposed design framework, the RobustRTO is a design level abstraction rather that its original concept [Lim00] of an implementation level structure. The RobustRTO concept is used to modularize the real-time system so that it is easier for the developer of the system to read the design. The concept of RobustRTO is similar to the packages in Java. Nonetheless defining a design-level specification for RobustRTO will serve as a good tool to the developer. Figure 6 describes the specification for the RobustRTO. The <MemberName> denotes the name of the underlying RTOs. <MetaObjectName> denotes the name of the underlying metaobject.

```
RobustRTO name {
        RTOMembers:
<MemberName>*
MetaObjects:
        <MetaObjectName>*
}
```

Figure 6. Specifications for the RobustRTO

3.4 Specification for Guards

Figure 7 shows the detailed specification for the Guards. <SubsystemName> denotes the name of the subsystem of the entire system that encapsulates the Guard. <RobustRTOName> denotes the names of the RTOs, which are to be monitored by the Guard. <Logname> denotes the log file name, which the Guard uses to determine the node failures. <ErrorActionName> denotes the type of error. Depending upon this type of error, the <ActionName> specifies the action to be taken.

```
Guard name {
        Subsystem:
                <SubsystemName>
        Members:
                <RobustRTOName>*
        Log:
                <LogName>*
                {<LogName> := <RobustRTOName><LogFileName>}
        ErrorAction:
                <ErrorActionName>*
                {<ErrorActionName> := <ErrorName><ActionName>}
                }
```

Figure 7. Specification for Guards

4 Prototype Implementation

A prototype of a simulated real-time system based on the proposed design framework was implemented. The prototype implementation was analyzed for the performance study of the overheads involved in the design. The prototype simulates a real-time defense system. There are four components of this real-time defense system: the radar subsystem, the central control subsystem, the decision subsystem, and the

action subsystem.

The radar subsystem monitors the external flying entities or entities approaching from the sea. The inputs from the command line simulate the radar subsystem. The radar subsystem passes its input to the central subsystem, which is the main control subsystem. This central control subsystem then invokes the decision subsystem to decide what action is to be taken for the identified external entities. Then the central control subsystem invokes the action subsystem to perform the specified action. Figure 8 describes these functional specifications for the prototype. The prototype involves around 1000 lines of Java code.

4.1 RTOs

The RTOs of the prototype are as follows:
CentralControl.java : is the main component of the prototype that runs continuously and accepts the input from the radar system.
DecisionSystem.java : contains the Message Triggered methods to decide on the input sent by the radar subsystem.
Action.java : contains the Message Triggered methods to take proper action.
ActionThread.java : is a thread that runs the Action subsystem. This is used for managing the primary and secondary Action threads. Partial code listing for this thread class is given below.

```
public class ActionThread extends Thread{

        // instance of Action
       private Action thisAct = null;

    …//some other variables

   // constructor initializes the thread variables
     public ActionThread(Action eAct) {
        setDaemon(true);
        thisAct = eAct;
     }

     public void run(){
       //invokes the current instance of the
      // of the action thread.
           thisAct.performAction();
            MetaAction.updateStatus(true);
    }
    …// other functions }
```

All the RTOs discussed above are stateless, in the sense that they do not maintain any database and they do not have to keep track of the system state. They are the so-called “control objects” in the object-oriented paradigm. At the design level, the timing constraints on the methods of these RTOs were made, but since the study was related to performance analysis the timing constraints had to be relaxed. These timing constraints could be useful if the prototype is ported to a distributed system, which can simulate the actual system. This is one of the future directions to the work described in the paper.

4.3 Self-stabilizing RTOs

DecisionSystem.java is the RTO, which is self-stabilizing. The decision subsystem could be qualified as a rule-based system. The modifications to make the decision subsystem self-stabilizing are indicated in the code. The Central control subsystem delegates the information to the decision system; hence it is not a rule-based subsystem and cannot be transformed into a self-stabilizing RTO.
A partial code listing of DecisionSystem.java is given below.

```
public class DecisionSystem{

        …  // get methods

        ….  // set methods

 // This is the main method of the class and contains self-stabilizing code
private void makeDecision(){
       if(this.enemy_sensor){
          if(this.latitude >= 25 && this.latitude <= 50){
            if(this.longitude >= 0 && this.longitude   <=
                                                    50){
                  ….        // actions involved } } }
// following code is added to make the function self-stabilizing according  to [Cheng92]
        else {
                  // initialize variables
                     Decision = false;
                    Weapon = null;
                     Pass = false;
                       }
                }
        }
```

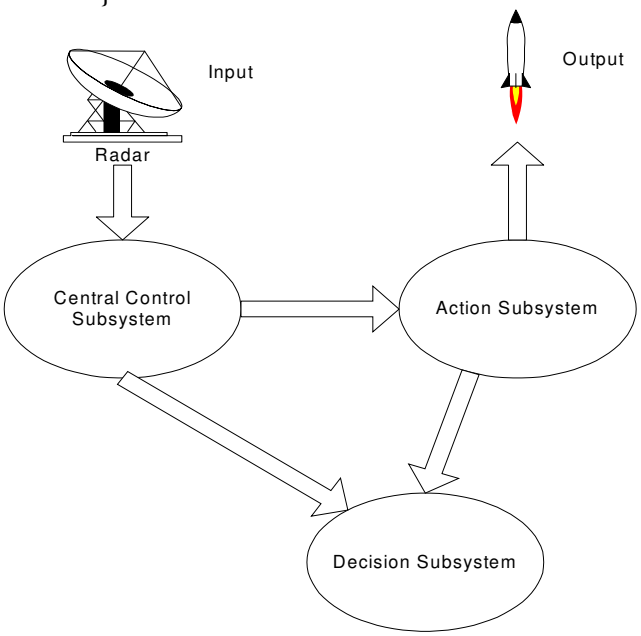

Figure 8. Functional specification for the prototype implementation

4.2. Metaobjects

The Metaobjects of the prototype are as follows:
MetaAction.java: encapsulates the Action RTO. The MetaAction metaobject implements the primary-backup replication [Guerraoui97] model. The MetaAction metaobject uses the ActionThread to create two threads as shown in figure 9.

The primary and the secondary threads are not concurrently running threads. The secondary thread runs only when the primary thread undergoes a nodal failure. All the subsystems are stateless and hence the primary and secondary threads need not communicate with each other for synchronization. Partial code listing of MetaAction.java is given below.

```
public class MetaAction{

    private SecondaryDecisionSystem secDec = new
                            SecondaryDecisionSystem();

    …. // all class variables

    public void performAction(){
       …. // some code goes here
       try {
          // the output of the action would be written in Guardlog incase of nodal failure
           bfw = new BufferedWriter(new
                    OutputStreamWriter(new

                FileOutputStream("Guardlog.txt")));
        }
        …. // some code goes here
       Thread primThread = new ActionThread(thisAction);
        primThread.start();

       ….. // code to check the first thread is here

       secThreadstart(); // start second thread if first fails
            }
```

MetaDecisionSystem.java: encapsulates the decision system RTO.

The main aim of the prototype is to establish a performance-based study. Hence the real-time property that is pertinent towards this study is the execution time. As a result the metaobjects of the prototype reflect only the execution time.

4.3. Guards

The action subsystem maintains a guard log that holds the information regarding the nodal failure of the subsystem. The only usage of the guard in the prototype would have been to signal that the nodal failure had occurred, hence the implementation of the guard is not considered important as far as the performance study is considered.

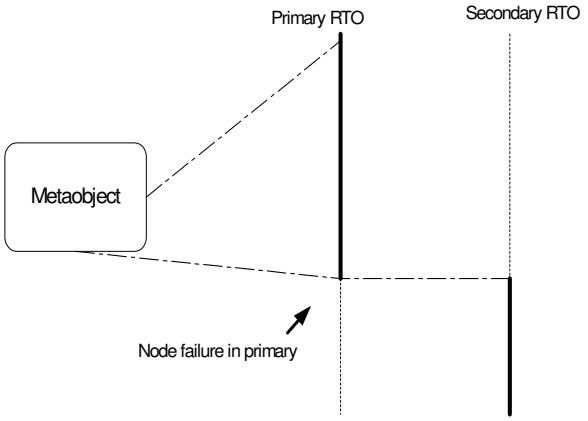

Figure 9. Action Thread implementation

4.4. Use of Inheritance

Inheritance could be used to accommodate the changes that need to be carried out at the runtime. In the prototype the primary and the secondary backup components of the decision subsystem RobustRTO use inheritance. The secondary RTO is inherited from the primary self-stabilizing RTO, the implementation of the secondary RTO is simpler than the primary RTO. The Action subsystem selects the secondary RTO at runtime if the primary self-stabilizing RTO undergoes a nodal failure. This is achieved through inheritance.

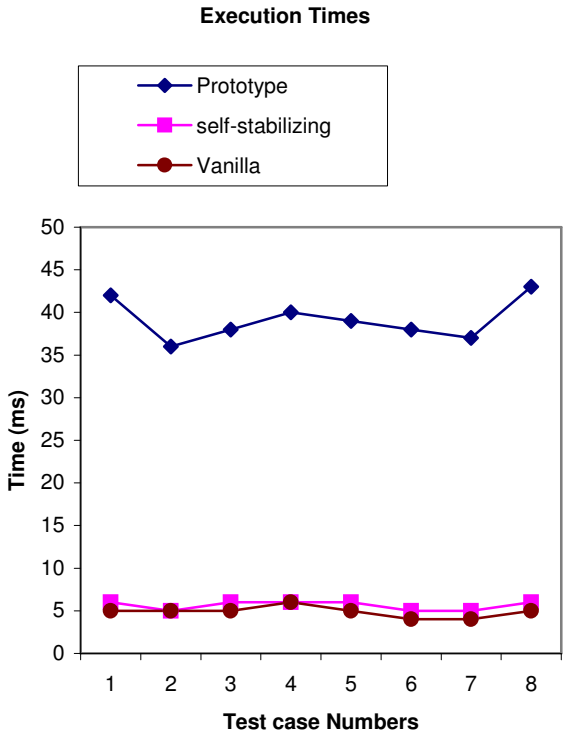

Figure 10 Performance chart

Vanilla (ms)	5	5	5	6	5	4	4	5
Self-stabilizing (ms)	6	5	6	6	6	5	5	6
Prototype (ms)	42	36	38	40	38	41	37	43

Table 1 Execution time data for each implementation
(ms denotes time in milliseconds)

5. Analysis of Implementation

The analysis of the implementation includes performance-based study. The performance of the prototype is based on the execution time of the modules. Then this performance is compared to the other two implementations. These implementations are described below:

Vanilla system: This implementation is very basic in the sense that it has no timing constraints, no self-stabilizing components, and the overall reliability is zero. There are no metaobjects.

Self-stabilizing system: This implementation contains self-stabilizing components only in addition to the Vanilla system. There are no metaobjects, no primary-secondary implementations and the overall reliability is achieved only through the self-stabilizing components.

All the three implementations are functionally similar and coded in Java, but the complexities of the implementations differ based on the degree of reliability that is achieved. To compare these three implementations, 8 test cases are identified. These test cases vary according to the input.

The testing was performed on the Themis SunOS 5.8 cluster of the department of Computer Science at the University of Houston. Due to huge and varying user activity on the cluster, proper time had to be selected at which the user activity is almost zero or almost constant. To reduce influence of other activities on the execution time, averages of 4 readings, taken at different time, have been used.

The performance chart shows that the vanilla implementation has the least execution time followed by the self-stabilizing implementation. The prototype has a very high overhead and the execution time is almost 5-6 times greater than the vanilla and the self-stabilizing implementations. The increased overhead in the prototype is a result of threading, inheritance, and primary-secondary implementation structure.

6 Conclusion

Self-stabilization is a very efficient way of implementing fault-tolerance mechanism for transient faults in a distributed system. It reduces the component redundancy. The proposed design framework to implement such a self-stabilizing system is very specific to the needs of the self-stabilizing system and allows the designer to focus on these needs. The basic element of the framework is the RTO which is further refined to self-stabilizing RTO if the possible.

Our analysis shows that the prototype implementation of the framework involves some overheads. By considering these overheads as a part of the WCET of the system modules one can meet the timing-constraints of the real-time systems. This framework may not be useful in situations where “short-execution-time” is more important than “on-time-execution”. Nonetheless, the design achieved through the proposed framework will aid the developer in visualizing the system before implementation.

In the past, many doubts have been raised over the use of object-oriented design in real-time systems. These objections [Healton98] suggest that object-oriented design is not a total solution for the problems in infrastructure deployment or software maintenance and moreover incurs additional overheads. The object-oriented paradigms like reuse, extensibility, etc., in a system could be achieved only if these needs are recognized at the design time.

As and when the self-stabilizing systems will evolve there would be many new things and needs that could be accommodated into the framework. The future work lies in deploying the prototype in a simulated or small scale distributed embedded systems. A scaled-up version of the self-stabilizing components could also

bring out some issues that could be solved at the design level and this is another direction of the future work. Currently, we are working to implement self-stabilizing system in Java and we hope to derive the new issues for the proposed design framework.

Bibliography

[Cheng92] Albert M. K. Cheng, "Self-Stabilizing Real-Time Rule-Based Systems", Proc. 11th IEEE-CS Symp. on Reliable Distributed Systems, Houston, Texas, pp. 172-179, Oct. 1992.

[Fabre97] Jean-Charles Fabre and Tanguy Perennou, "A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach", IEEE Transactions on Computers, Vol 47, pp. 78-95.

[Guerraoui97] Rachid Guerraoui and Adre Schiper, "Software-Based Replication for Fault Tolerance", IEEE Transactions on Computers, vol.30, no.4, pp68-74, 1997.

[Healton98] Bruce Healton, Arnold Kwong and Rodney Lancaster, "Objection to Objects". IEEE Object-Oriented Real-Time Distributed Computing, pp. 95-104, 1998.

[Kalbarczyk99] Z.T.Kalbarczyk, R.K.Iyer and et al., "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance", IEEE Transactions on Parallel and Distributed Systems, vol.10, No.6, pp. 560-579, 1999.

[Kim97] K.H.Kim, "Object Structures for Real-Time Systems and Simulators", IEEE Computer, pp. 62-70, Aug 1997.

[Kim98] K. H. Kim, "ROAFTS: A Middleware Architecture for Real-time Object-oriented Adaptive Fault Tolerance Support", Proc. Of High Assurance Systems Eng. Symp., pp.50-57, 1998.

[Lim00] Hyung-Taek Lim, Seung-Min Yang, "A Framework to Model Dependable Real-Time Systems based on Real-Time Object Model", Proceedings of 7th International Conference on Real-Time Computing Systems and Applications, pp. 31-38, 2000.

[Mitchell97] S. E. Mitchell, A. Burns and A. J. Wellings, "Developing a Real-Time Metaobject Protocol", IEEE Computers, pp. 323-330, 1997.