Architecture Support for Behavior-based Adaptive Checkpointing

Nianen Chen, Shangping Ren
Illinois Institute of Technology, Chicago, IL, U.S.A.
Email: {nchen3, ren}@iit.edu

Abstract—**Checkpointing is a commonly used approach to provide system fault-tolerance. However, using a constant checkpointing frequency may compromise the system's overall performance when there are multiple types of QoS requirements involved. Hence, it is important that the checkpointing frequency is customizable and runtime adaptable. However, for open distributed and embedded applications, often there is a large number of entities involved in an application and these entities may join or leave the system frequently. The scale and the dynamicity make it difficult to apply the adaptive checkpointing strategy unless we have a model that encapsulates the issues within a well-defined structure and further shields complexity from application developers. In this paper, we introduce a behavior-based adaptive checkpointing approach for open systems and present an architecture support to optimize system's overall performance through using adaptive checkpointing frequencies.**

Index Terms—**behavior-based adaptive checkpointing, role-based coordination, deadline, fault-tolerance, dependability, availability**

I. Introduction

Open distributed and embedded (ODE) systems often have multiple Quality-of-Service (QoS) requirements. The overall performance of an ODE system is usually judged by the integration of multiple QoS measurements, such as system availability (the average ratio that a system can stay at the operational state during its life-time), task deadline miss probability (the probability that a task may violate its timing constraint), and task execution time.

Checkpointing Rollback and Recovery (CRR) uses temporal redundancy to achieve fault tolerance [1]. Previous researches have shown that checkpoint frequencies can impact system availability [2], task deadline miss probability [3], and task execution time [4]. Clearly, taking more frequent checkpoints speeds up system recovery when failures occur, and hence may improve the system's availability, accelerate its respond time, and reduce the probability of missing deadlines in the presence of faults. However, checkpointing also takes time and consumes resources. It hence increases fault-free execution time and may jeopardize the satisfaction of timing constraints. Therefore the checkpoint interval needs to be carefully analyzed and adaptive as it decides system's overall performance.

However, modeling and specifying the rules for adaptive checkpointing is difficult in ODE systems. It is because there is no closed system boundary in this type of systems --- embedded entities may join and leave the system freely. Furthermore, the scale for such systems is often very large. Applying adaptive checkpointing on these large amount and dynamic entities is an even more challenging task.

On the other hand, although an ODE system may contain large number of dynamic embedded entities, these entities often share behaviors. A sensor monitoring system shown in Figure 1 is an example.

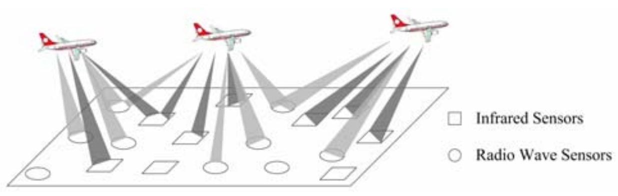

Figure 1. Sensor Monitoring System

In the system, there can be large number of infrared and radio wave sensors distributed in the field to monitor the space. These sensors may be active or inactive depending on where a flying object occurs. From behavior perspective, there are only two types of behaviors, i.e., infrared sensoring and radio wave sensoring. Hence, if we use behavior to abstract embedded entities, we not only reduce system scale, but also free ourselves from considering individual entities and its dynamicity while handling constraint specifications and realizations.

Based on the above observation, the Actor Role Coordinator (ARC) model is developed for modeling ODE systems [5]. With the ARC model, an ODE application's functional logic is modeled as actor computations. The actor's "behavior" is abstracted by its corresponding role. The QoS constraints and the adaptive checkpointing rules are defined based on behaviors instead of the individual actors. The role's behavior abstraction hence decouples the syntactic dependencies between the coordinators and the actors, thus shields the coordinator layer from the dynamicity and large number of underlying actors.

In this paper, we present an architecture support for the ARC model and for achieving the behavior-based adaptive checkpointing in an ODE system. It is a layered structure and contains two layers. The Actor layer implements the ARC model and provides an interface for creating actors, roles and coordinators. In addition, this layer is also responsible for monitoring actor behavior change and deciding actor memberships, for event propagation and also for imposing QoS constraints on actors by managing messages targeted at the actors. The QoS layer bridges the high-level ARC model and behavior-based constraint specification with the underlying physical systems by implementing multiple service components.

This layered architecture allows the ARC layer to focus only on the specification and management of group-based coordination constraints without caring the details of applying them on individual actors. QoS layer is a bridge between the user level role-based adaptive checkpointing constraints and the system level actor-based checkpoint interval values. The adaptation decisions and adaptation rules are passed from the ARC layer to the QoS layer, where optimal checkpoint intervals are computed and applied on the affected actors. Through the cooperation of the ARC and QoS layers, behavior-based adaptive checkpointing can be achieved systematically. The resulting architecture is more flexible, reusable, and evolvable.

The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the ARC model and illustrates how it facilitates behavior-based checkpointing. Section IV describes the architecture support for adaptive behavior-based checkpointing and presents the detailed design of the ARC layer and the QoS layer. Finally, Section V draws conclusion and points out our future work.

II. Related Work

There is a large research community aiming to facilitate the specification and enforcement of QoS requirements in a systematic way. For instance, CORBA middleware technology [8] provides applications with a set of standard services, such as fault tolerance, real time scheduling and resource control services. However, in CORBA, fault tolerance and timing requirements are handled separately as different services. No mechanism is provided to compose and coordinate these two requirements in a conjunctive way.

The QuO framework [10] supports adaptive QoS for objects in distributed applications. In such a framework, QoS requirements are specified separately with a special designed QoS declarative language, and maintained in an entity named “contract”. Through run-time monitoring of system conditions, the contract can be changed to adapt to the dynamic running environment. However, the QuO lacks of specific solution for an application to adaptively decide its checkpoint interval that will optimize system overall performance measures, such as the integrated measure of system availability, deadline miss probability and task execution. Users have to develop their own ad-hoc mechanisms to apply their adaptive checkpointing rules, which is error prone.

A mode-driven architecture is proposed in [11] to provide various levels of dependability based on an application’s modes. A mode is defined as a system behavior of an application under a specific system condition. This framework provides architecture supports for mode specification, mode-change detection, and resource monitoring so that different dependability requirements can be applied on multiple application behaviors. The mode concept is quite similar to the behavior abstraction that we have adopted in our paper. However, the model-driven architecture itself does not provide a systematic solution to help users identify the relation between checkpointing frequency and different QoS requirements, and its impact on system overall performances.

In [9] we built a framework to realize the ARC model. The framework focuses on dealing with the behavior-based constraints specifications. The adaptive checkpointing itself is not part of the framework. Application users still have to implement their adaptation rules and algorithms. In this paper, we extend the framework by introducing an extra QoS layer to integrate the adaptive checkpoint mechanisms into the framework. We explicitly define the checkpointing adaptation goal as to maximize system overall performance matrix, which may include system availability, task deadline miss probability, and task execution time. We formulate the optimal checkpointing interval problem as a Mixed Integer Non-linear Programming problem. Services in QoS layer calculates and then enforces optimal checkpoint interval without user intervening.

There are researches in the area of optimizing the checkpoint interval to maximize system availability [13, 14], minimize response time [4], or minimize deadline miss probability [5] separately, however few of them address these QoS constraints in conjunction. For ODE systems where multiple QoS requirements co-exit and compete for limited resources, these models and analysis fall short to generate adaptive strategies that guarantee optimal overall performance.

III. The ARC model

The ARC (Actors, Roles and Coordinators) model is developed to address the dynamicity, large scale and existences of multiple types of QoS requirements in open distributed and embedded systems. Conceptually, the ARC model is a composition of three layers, with each of the three components associated with a dedicated layer as shown in Figure 2.

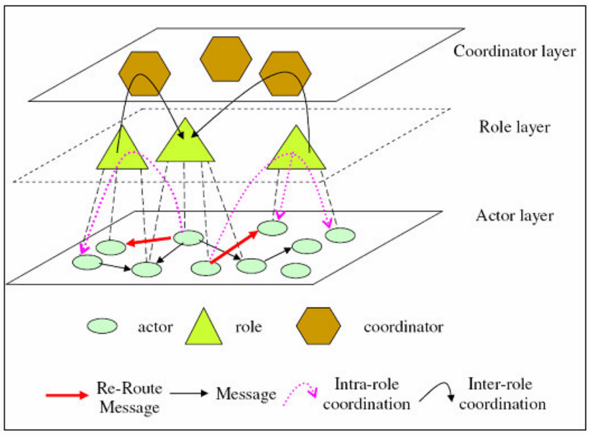

Figure 2. The Actor, Role, Coordinator Model

The separation of concerns is apparent in the relationships involving the layers. The actor layer is dedicated to functional behavior and is transparent to the coordination enacted in the role and coordinator layers. The roles and coordinators constitute the coordination layer responsible for specifying and imposing coordination constraints among the actors. The coordinator layer is oblivious to the actor layer and is reserved for role-based (inter-role) coordination. The role layer bridges the actor layer and the coordinator layer and may therefore be viewed from two perspectives. From the perspective of a coordinator, a role enables the coordination of a set of actors that share the same static description of behaviors without requiring the coordinator to be aware of the individual actors in the set. From the perspective of an actor, the role is an active coordinator that imposes intra-role coordination constraints on messages sent/received by the actor. More detail discussion about the model can be found in [5].

We use the ARC model to facilitate the specification of behavior-based constraints that are used to adaptively select optimal checkpoint intervals at runtime. These constraints may include the timing constraints, the system availability constraints, and the deadline miss probability constraints. In our system, a behavior is defined as a tuple <*O, A*>, where *O* is a set of operations and *A* is a set of attributes that a computation entity can perform and process. For instance, a Video-On-Demand (VOD) server actor's behavior can be described as <*Play_Video, Property_From*>. It will hence exhibit different behaviors when it accepts service demands from different types of clients. For example, if it is handling a request from a "VOD Guest Client" actor, it exhibits a <*Play_Video, "VOD Guest Client"*> behavior; and its behavior switches to <*Play_Video, "VOD VIP Client"*> when it is processing a request from a "VOD VIP Client" actor. These two behaviors are abstracted into "VOD_VIP_Server Role" and "VOD_Guest_Server Role", respectively. Instead of defining constraints and applying checkpointing interval on each VOD Server actor individually, using ARC model we only need to specify two sets of constraints on the above two roles, compute two checkpoint intervals, and apply the values on all actors that share the same behaviors.

In the ARC model, rules to calculate and enforce checkpoint interval are declared based on roles and stored in coordinators as inter-role coordination constraints. Roles and coordinators are treated as special coordination actors that are able to handle specific types of messages, i.e. events --- certain state changes in the system. Upon observing an event, the coordinators retrieve eligible adaptive checkpointing constraint depending on the current application behavior. The behavior-based checkpointing constraint will then be propagated to the corresponding role where it is transferred into an actor-based (intra-role) coordination constraint and enforced on the individual actors.

IV. ACHIEVING BEAHAVIOR-BASED ADAPTIVE CHECKPOINTING THROUGH THE ARC MODEL

Our goal is to build a framework to specify behavior-based checkpointing constraints, calculate corresponding checkpoint interval that optimizes system overall performance, and enforce the checkpointing interval on computation entities. To fulfill these tasks, we first present the main design criteria and then present our solution in detail.

A. Design Goals

The main design and implementation concern of the framework is to provide a level of abstraction that implements the behavior-based adaptive checkpointing, and at the same time provide good performance, scalability and flexibility for different applications.

User interfaces. From an end user's perspective, the adaptive checkpointing framework must have interfaces to allow them specifying behavior-based QoS requirements and their integrations, and algorithms for computing optimal checkpoint intervals.

Performance and scalability. The introduction of active roles in the ARC model helps mitigate the scalability issues in coordination management by allowing coordinators to only coordinate roles, while roles only coordinator actors who share the same behaviors. However, since the coordination in the ARC is transparently enforced on underlying actors, when the number of actors increases, two problems may emerge: (1) Every coordinated message triggers at least one event that is required to be handled by remote coordination actors, hence the communication overheads may be high; (2) Roles and coordinators become potential bottlenecks, which may degrade performance and make systems hard to scale. To alleviate the problems, we have developed a decentralized architecture to further distribute coordination behaviors and states to local physical nodes to avoid communication overhead.

Coordination behavior reusability. Coordination constraints are high level rules that should be reusable in a variety of scenarios. The detail mechanisms to interact with the middleware and system level services, to intercept and encapsulate messages, to calculate optimal checkpoint intervals, and to collect and validate user specifications, should be decoupled from the high level coordination behaviors.

Mechanisms for deciding optimal checkpoint interval. No matter which optimal checkpoint interval selection algorithm is adopted, the framework needs to provide mechanisms to compute optimal checkpoint interval based on the current available resources, system parameters, and user defined behavior-based QoS requirements. The framework has to further enforce the checkpoint interval on the actor that is responsible for performing the task.

The rest of the section will discuss the detailed design.

B. Architecture Overview

Our framework is based on layered structure. As shown in Figure 3, the top layer is the ARC layer that realizes the ARC model semantics. The detail design of the layer will be given in subsection *C*. To decouple the high level coordination behaviors from the middleware level implementation details, we provide another level of abstraction to bridge the ARC layer and middleware common service layer. We call this layer the “QoS layer”.

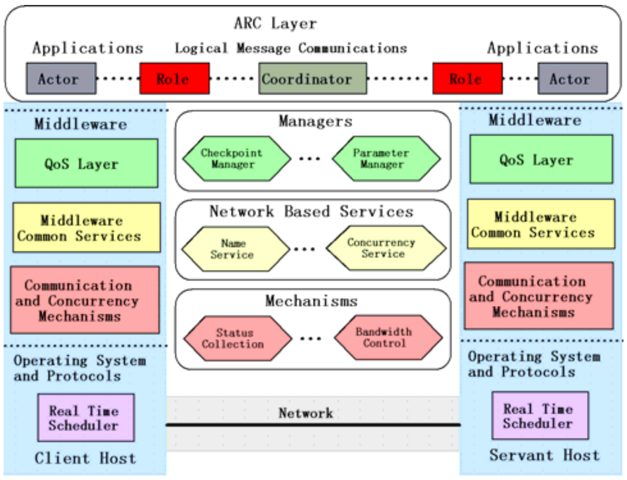

Figure 3. Adaptive Checkpointing Framework Structure

The main components provided in the QoS layer are: *Message Manager, User Interface Manager, Optimization Manager, Parameter Manager,* and *Checkpointing Manager*. These managers are to achieve adaptive checkpointing functionalities. The detail designs of the QoS layer will be presented in subsection *D*.

C. ARC Layer

As we have discussed, the ARC layer implements the ARC model and provides an interface for creating actors, roles and coordinators and for managing behavior-based coordination constraints. In addition, the layer is also responsible for monitoring actor behavior change and deciding actor membership, event propagation and imposing QoS constraints on actors by managing messages targeted at actors. Figure 4 depicts the architecture of the ARC layer. It’s worth noting that we use a concrete middleware technology, i.e. the CORBA, in our prototype implementation.

Both computation actors and coordination actors (i.e., roles and coordinators) are implemented based on Actor semantics defined in [18], where actors are autonomous and active objects that communicate with each other through asynchronous messages. Each actor has three primitive operations: *send, rcv, and new*, to send and receive messages, and create new actors.

In the ARC layer, an Actor Platform is installed on every physical node. An Actor Platform is implemented as a “system actor”, which creates actors, roles, and coordinators, initializes their states and behaviors, sends messages and generates events. Accompanied with each actor creation (including both computation and coordination actors), the Actor Platform also creates a Message Manager object to handle actor communication tasks. When an actor tries to send a message to another actor, it delegates the message to its Message Manager. For sender actor, the Message Manager acts as a CORBA client object to asynchronously send the message to destination actor’s message manager, which acts as a CORBA service object. The receiving message manager further forwards the message to the receiving actor for processing. Hence, the CORBA middleware details are encapsulated in the implementation of message manager and are transparent to application developers.

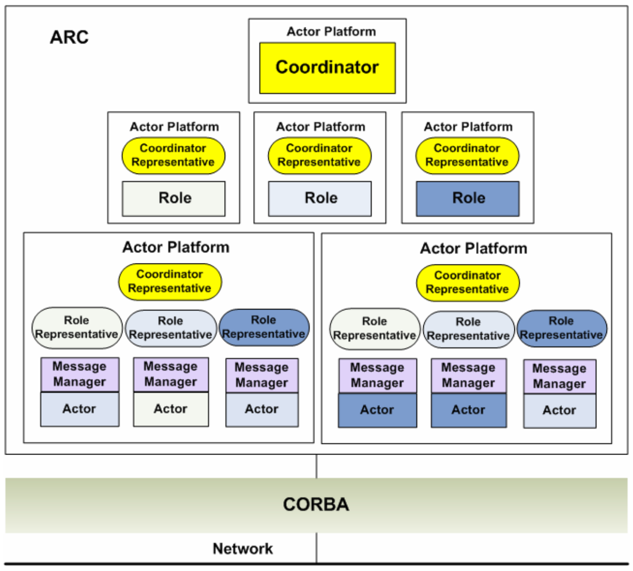

Figure 4. The ARC Services Structure

We define two supporting entities: Coordinator Representative (CR) and Role Representative (RR). As the name suggested, they represent coordinators and roles to perform coordination behaviors in local Actor Platforms. These representatives are implemented as coordination-actors. According to the definitions of coordination actors, they are able to communicate with each other through event communications.

The local Actor Platform creates a Role Representative coordination actor for every existing role to fulfill both its membership management behavior and coordination behavior. In the ARC model, it is the roles, but not the actors, who manage the group membership. More specifically, when a new actor is created or an actor changes its behavior, the roles compare the actor behavior description with their membership criteria. Qualified actors will be added into an actor-role association structure cached in each local role representative.

In the ARC service layer, coordination constraints are transparently applied on actors. It is achieved through buffering the messages in receiver actors’ mailboxes by Message Managers, obtaining coordination constraints through forwarding events to corresponding role representatives and coordinator representative for constraint checks, and then applying such coordination constraints by manipulating the messages in the mailboxes.

The Message Manager is responsible for encapsulating the original message into an event and forwarding it to the corresponding coordination actors for processing.

Events in our system are special messages containing two sets of information, i.e., regular operational information and coordination information. Coordination information contains the original message and the coordination information describes the players who are involved in such a message communication. The player information is obtained by checking the actor-role association information.

When the Coordinator Representative observes the propagated event, it searches its constraint store to locate proper constraints based on the coordination information. The constraint is then forwarded to corresponding Role Representative for enactment. As all these operations are performed locally and no remote communications are required, the constraint propagations do not introduce much performance overhead.

In order to evaluate the performance overhead, we have developed a prototype of the ARC service layer. The experiment settings are as following: two Intel x86 machines. The first machine is a Pentium IV 1.7 GHz with 512MB RAM and the second is a Pentium IV 3.06GHz with 1GB RAM. Both of them are running on Windows XP and connect with each other through a 100M Ethernet switch. In our experiments, we develop a simple Ping Pong application, which asks two actors, i.e. the Ping actor and the Pong actor, in different machines to continuously send and reply a specific number of messages to each other.

Figure 5 demonstrates the situation when multiple actor pairs run and send messages concurrently. For example, there are 100 actors run on one machine, and 100 actors run on another machine. These 100 pairs of actor send and reply messages to each other concurrently. As the figure shows, the increases of number of actors bring little impact on the performance of the ARC.

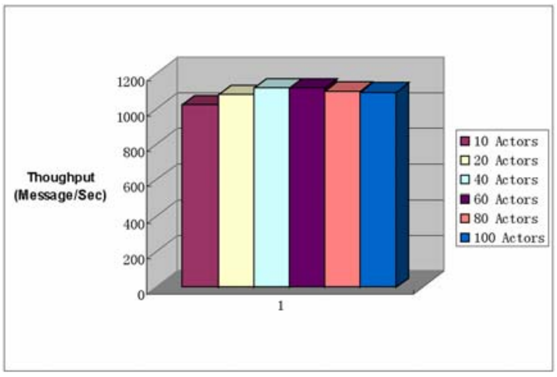

Figure 5. Performance of ARC when the number of actors increases

We further test the overhead brought by introducing the role coordination entities to achieve intra-role coordination constraints. There are 10,000 messages sent between two actors on different machines. We introduce two roles, i.e. the PingRole and the PongRole, and divide the constraints into three categories: 20% inter-role constraints stored in a coordinator, 40% intra-role constraints stored in PingRole, and another 40% intra-role constraints stored in PongRole. In those constraints, we perform simple string comparisons to simulate the constraint checks. Figure 6 gives the measurements.

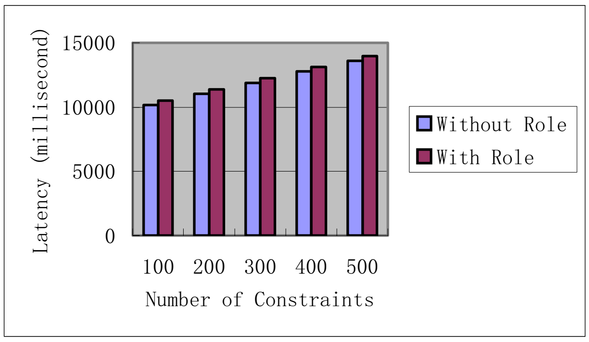

Figure 6. Overhead of introducing role layer

As shown in Figure 6, when there are 100 constraints in a single coordinator, the overhead of introducing two extra role entities is about 3.5%; when there are 500 constraints, the overhead is about 2.7%. The main overhead comes from two extra communications between the sender and receiver actors and their attached roles, which is fixed no matter how many constraints need to be checked. When there are no roles, a coordinator has to check all these constraints; in contrast when there are two extra roles, the coordinator only handle 20% of the constraints, and rest of the constraints are distributed to roles for processing. Therefore the overhead actually decreases when the number of constraints increases.

D. QoS Layer

As discussed in subsection *B*, the QoS layer consists of five managers.

Message Manager. The message manager provides mechanisms to intercepts messages sent between actors for roles and coordinators to enforce coordination constraints, and hide network level message communications from ARC layer.

User Interface Manager. This manager provides a set of interfaces for specifying behavior-based QoS requirements, composition functions, and optimal checkpoint interval algorithm. An example of using the interfaces will be given in our case study in the next subsection.

Optimization Manager. Based on the individual QoS requirements, QoS composition functions and the checkpoint interval calculation methods specified by users, the optimization manager decides a checkpoint interval that optimizes the overall system performance.

Checkpointing Manager. This manager is responsible for enforcing the derived checkpointing interval on an actor. Specifically, the manager provides Logging and Recovery Management Mechanisms that support the system-controlled checkpoint rollback recovery. During normal operation, the logging mechanism takes checkpoints and Logs Messages to record actor states. After a fault, the Recovery Mechanism retrieves these records from the log and uses them to restore the state.

The Checkpointing Manager also provides a Property Configuration Interface to accept the interval decided by the optimization manager. All actors that need checkpointing service are required to implement such an interface. Base on the specified checkpoint interval, the Logging mechanism in the manager periodically obtain the state of the object. During recovery, the Recovery Mechanism sets its state to the most recent state that was recorded in the log. All of these operations are automatically performed by the framework and are transparent to the computation actors.

Parameter Manager. There are a few key system parameters that the Optimization Manager needs in order to calculate the optimal checkpoint interval (OCPI). Since we use CRR mechanism to provide fault recovery, the related costs include the time to place a checkpoint and log a message, the task worst case execution time, the time to roll-back to the previous checkpoint in the presence of faults, and the time to replay a message. The parameter manager relies on two helper objects to provide these data, i.e., the Resource Monitor, and the Execution Time Analyzer. The open source resource monitor tool named StatSentry [15] can be used to monitor resource usage and collect runtime system runtime information. The execution time analyzer utilizes an execution time analysis tool (the Bound-T analysis tool [16] in our implementation).

Managers in QoS layer are special entities in our framework. On one hand, to maintain both syntax and semantics consistency in ARC level, the managers have to play as actors when communicate with computation actors, roles and coordinators. On the other hand, they need to interact with middleware common services to help roles and coordinators fulfill coordination behaviors. Therefore they have to act as middleware objects. Figure 7 demonstrates the two interfaces that a QoS layer manager has to implement.

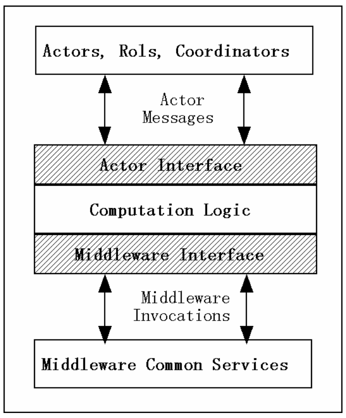

Figure 7. Manager Abstract Structure

Figure 8 depicts the architecture of the behavior-based adaptive checkpointing framework. The roles and coordinators in the ARC layer control the flow of the coordination constraints in an application level. QoS layer managers are responsible for handling the enforcement details in middleware level. The detail workflow of the framework will be explained in a case study in section E.

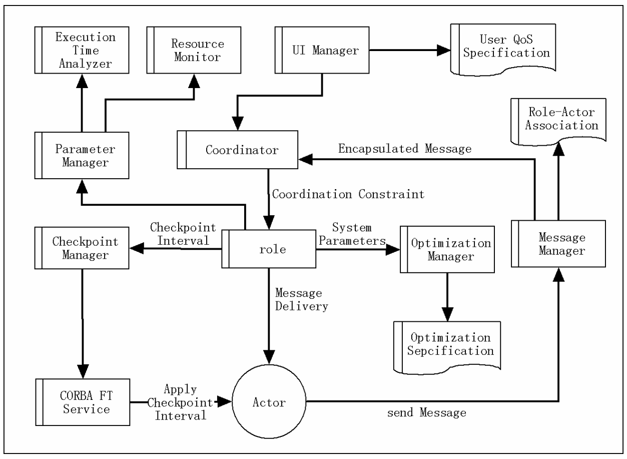

Figure 8. Adaptive Checkpointing Framework Architecture

E. Workflow of the Framework – A Case Study

In this section, we use the VOD example described in section III as a case study to explain the workflow of how the framework helps to achieve behavior-based adaptive checkpointing and optimize the system's overall performance.

Each time when a message representing a service request sent from a VOD client to a VOD server, the checkpoint interval will be adaptively decided to optimize the VOD server's overall performance. The constraints that facilitate generating the runtime optimal checkpoint interval are specified based on different VOD server roles.

Currently, we use a simple additive weighted value to represent the system overall performance, i.e.,

$$score = \omega_A N_A + \omega_T N_T + \omega_R N_R \quad (4.1)$$

where N_A, N_T, N_R are normalized values for system availability, task deadline miss probability, and task execution time, respectively, and $\omega_A, \omega_T, \omega_R$ are their corresponding weight values ($\sum_{i=A,T,R} \omega_i$ =1) that are specified by the user. To optimize system overall performance is to maximize the *score*. The N_A, N_T, N_R are determined by A_{low} , T_{up} and $[R_{low}, R_{up}]$, respectively, where A_{low} is the lower bound of system availability that a user can accept; T_{up} is the upper bound of task deadline miss probability that a user can accept; $[R_{low}, R_{up}]$ is a range of task execution time, where R_{low} is the execution time in an best case environment where no fault will occur and no fault tolerance is necessary, and R_{up} is the upper bound of task execution time that a user can accept.

As an example, consider the following scenario for which the guest client requires that the VOD system must be at least 90% available, must finish the play within [6, 15] time frame, and the deadline miss probability for each frame cannot be larger than 30%, the importance for these criteria is 0.5, 0.25 and 0.25, respectively. The VIP Server role has different requirements as shown in table 1.

	A (%)	R (s)	T (%)	W
Guest	90	[6,15]	30	(1/2,1/4,1/4)
VIP	99	[6,10]	10	(1/4,1/4,1/2)

Table 1. Role-based QoS Specifications

The normalized values, N_A, N_T, N_R, is calculated as following:

$$N_A = \frac{A - A_{low}}{1 - A_{low}},\ N_R = \frac{R_{up} - R}{R_{up} - R_{low}},\ N_T = \frac{1-T}{1-T_{up}} \quad (4.2)$$

The Parameter Manager shall also collect the following constants that will be used to solve the optimization problem:

C: time to take a checkpoint; *L*: time to log a message; α : time to recovery the system state from the most recent checkpoint; β : time to replay a message; λ : fault arrival rate; γ: message arrival rate; *W*(0): worst case execution time without considering faults. To facilitate our case study, we assume a sample list of these parameters given in Table 2:

C	L	α	β	λ	γ	$W(0)$
0.2	0.1	0.1	0.1	0.005	0.01	6

Table 2. Role-based QoS Specifications (time unit: second)

Based on our study [17], the relationship between the checkpointing interval *Y* and *A, T, R* are as following:

$$(1)\ A = Y/(C + (\gamma + L + \lambda\beta + 1)Y + \lambda\alpha\gamma Y^2/2)$$

$$(2)\ T = 1 - e^{-\lambda}\sum_{m=0}^{k}\frac{\lambda^m}{m!}$$

$$(3)\ R = W(0) + (W(0)/Y - 1)C + \gamma L W(0) + k(\alpha\gamma Y + \beta)$$

$$(4)\ k = \left\lfloor \left(\frac{\sqrt{\gamma\alpha C W(0) + (R - W(0) - \gamma W(0)L + C)\beta} - \sqrt{\gamma\alpha C R}}{\beta} \right)^2 \right\rfloor$$

where *k* represents the number of faults that can be tolerated, which is determined by the *A, T* and *R*.

From the constraints, the *A, T*, and *R* are all non-linear functions of *Y*. Meanwhile, the above problem has integer variables (the *k*) and linear objective function. The optimization problem is hence categorized as the Mixed Integer Non-linear Programming (MINLP) problem and the optimal solution of *Y* for this problem maximizes the overall system performance.

Based on the information given in table 1, table 2, the objection function defined in (4.1), and the MINLP problem defined above, we follow the numerical analysis method defined in [17] and obtain the following results depicted in Figure 9:

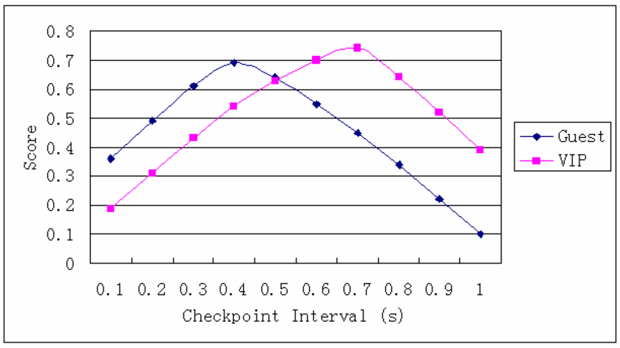

Figure 9. Checkpoint Interval vs. Performance Score

From the Figure, we can see that the optimal checkpoint intervals is 0.4 (score is 0.69) for actors with VOD Guest Server Role and 0.7 (score is 0.77) for actors with VOD VIP Server Role, which means that checkpoints will be taken every 0.4 second and 0.7 second for two types of roles respectively to maximize their performance scores.

We finally use the "VOD VIP Server Role" as an example to demonstrate the procedure of computing and applying an optimal checkpoint interval. After the role-based QoS constraints are specified, they will be "pushed" by the User Interface Manager to the coordinator constraint store. Specifically, after the specification is submitted, the following constraint depicted in Figure 10 will be added into the coordinator constraint store:

```
DECLARATIONS
    EVENT event;
    ROLE role;
 RULES
    IF (event.EventTypeName = = "SendMessage")
    {
      role = event.ReceiverRole;
      IF ((event.SenderRoleTypeName = = "VIPVODClientRole")
        AND (event.ReceiverRoleTypeName = = "VIPVODServerRole"))
             SendEvent (role, tell, "set availability = [0.99, 1],
                           set task_deadline_miss_probability = [0, 0.1],
                           set task_execution_time = [6,10]");
    }
```

Figure 10. Inter-role coordination constraint

When a message is sent from an actor associated with the "VIP VOD client Role" to a destination actor with "VIP VOD Server Role", the workflow to apply a behavior-based checkpointing can be described as follows:

(1) The message is intercepted by the Message Manager and encapsulated as an event.
(2) This event is observed by the coordinator and triggers the constraint depicted in Figure 10.
(3) By interpreting the constraint, the coordinator forwards the user specified QoS criteria to the "VOD VIP Server Role".

(4) On receiving the propagated QoS constraints, the VOD VIP Server Role sends a message to the Parameter Manager to retrieve the system parameters.
(5) Upon receiving the system parameters, the server role sends another message attached with the QoS constraints and system parameters to the Optimization Manager to ask for optimal checkpoint interval.
(6) The optimization manager uses the aggregation function, the optimal checkpoint interval calculation algorithm (MINLP), user specified QoS requirements and the system parameters to decide the optimal checkpoint interval .
(7) After receiving the checkpoint interval, the VOD VIP Server Role forwards the value to the Checkpoint Manager, where the checkpointing constraint is finally enforced on the VOD serve actor that will perform the task execution.
(8) The Checkpointing Manager sends a message to the VOD server role to notify the successful enforcement of the checkpoint interval, and the role then release the message to the VOD server actor for execution.

V. Conclusions and Future Work

In this paper, adaptive checkpointing is proposed to be applied on ODE systems to achieve optimal performance where the performance matrix is composed of system availability, task deadline miss probability, and task execution time. To reduce the complexity of applying the adaptive checkpinting, we introduce an Actor, Role, Coordinator (ARC) coordination model to model the ODE system computations and treat behavior-based adaptive checkpointing constraints as role-based coordination rules. Individual entity's behaviors are abstracted into different roles. The checkpoint interval settings on these entities are defined on the roles they are playing, which is more stable and more scalable.

The behavior adaptive checkpointing framework is hence divided into two layers: (1) An ARC layer that realizes the actor, role and coordinator entities and their interactions. Our experiment results show that the overhead caused by introducing extra coordination entities, i.e., roles and coordinators, is limited and the layer scales well when more coordination constraints or more functional units are involved. (2) To separate the adaptive checkpointing specification and implementation concerns, we introduced a QoS layer between the ARC layer and the middleware common service layer. This layer provides services to systematically monitor resources, compute optimal checkpoint interval, and enforce the interval on actors. Applications programmed in ARC layer can simply use these services to achieve adaptive checkpointing without concerning the implementation details.

Our future work is to produce a prototype of the QoS layer, integrate it with the existing ARC prototype [9] into a complete behavior based adaptive checkpointing implementation. Experiments will then be run on this integrated prototype to evaluate how the adaptive and behavior-based specifications help improve overall system performance, and measure the performance, overhead and scalability of the framework.

References

[1] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys, Vol. 34, No. 3, September, 2002. Page 375-408.
[2] E. Gelenbe, D. Derochette. Performance of Rollback Recovery Systems under Intermittent Failures. Communication of the ACM. Volume 21. 1978.
[3] H. Lee, H. Shin and S. Min. Worst case timing requirement of real-time tasks with time redundancy. In Proc. Real-Time Computing Systems and Application. 1999. 410-414.
[4] S. W. Kwak, B. J. Choi and B. K. Kim. An optimal checkpointing-strategy for real-time control systems under transient faults. In IEEE Transaction of Reliability, vol. 50, no. 3, pp. 293-301, 2001.
[5] S. P. Ren, Y. Yu, N. E. Chen, M. Kevin, and L. M. Shen, P. Pierre. Actors, Roles and Coordinators — A Coordination Model for Dynamic Distributed Open Systems. In *Proc. of 8th Conference on Coordination Models and Languages*, 2006.
[6] C. L. Hwang and K. Yoon, Multiple Criteria Decision Making, Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 1981.
[7] D. C. Schmidt et al. The design of the TAO real-time Object Request Broker. In Computer Communications, 1998.
[8] Object Management Group. CORBA 3 specification, In OMG Technical committee Document, 2002.
[9] N. E. Chen, S. P. Ren. Building a Coordination Framework to Support Behavior-Based Adaptive Checkpointing for Open Distributed Embedded Systems. HICSS 40th Annual Hawaii International Conference. 2007.
[10] J. A. Zinkey, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. In Theory and Practice of Object Systems, 3(1), 1997.
[11] D. Srivastava and P. Narasimhan. Architectural Support for Mode-Driven Fault Tolerance in Distributed Applications. In Proc. of ICSE 2005 Workshop on Architecting Dependable Systems, 2005.
[12] J. W. Young. A First-Order Approximation to the Optimum Checkpoint Interval. Communication of ACM. 530-531. 1974.
[13] K. M. Chandy, J. C. Browne, C. W. Dissly, W. R. Uhrig. Analytic Models for Rollback and Recovery Strategies in Database Systems. IEEE Transaction of Software Engineering. SE-1, 1, 100-110. 1975.
[14] K. M. Chandy. A survey of analytic models of rollback and recovery strategies. Computer 8, 5. 40-47. 1975.
[15] Resource Statistic Monitor – StatSentry. http://resourcemntrd.sourceforge.net/
[16] "Using a WCET Analysis Tool in Real-Time Systems Education" Euromicro Worst-Case Execution Time Workshop 2005 (WCET 2005)
[17] N. E. Chen, S. P. Ren. Performance Optimization of Message Logging based Rollback Recovery in Distributed Real-time Embedded Systems. Technical Report. http://sunrise.cs.iit.edu/chen_1070819.pdf
[18] Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1986.