Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

IMPLEMENTATION OF TIME MANAGEMENT IN A RUNTIME INFRASTRUCTURE

Buquan Liu
Yiping Yao
Jing Tao
Huaimin Wang

School of Computer
National University of Defense Technology
Changsha, Hunan 410073, CHINA

ABSTRACT

The High Level Architecture (HLA) time management is concerned with mechanisms for guaranteeing message order, process synchronization and execution correctness in distributed simulations. Time management greatly influences on the scales of applications, especially for the computation of Greatest Available Logical Time (GALT) and the implementation of optimistic services. StarLink is an RTI with central architecture, which is compliant with the IEEE 1516 standard. This paper systematically describes the implementation algorithms for main time management services in StarLink. Two smart and efficient algorithms about GALT computation and optimistic services are also introduced, which are suitable for many RTIs such as RTI1.3-NG, pRTI and DRTI. For the GALT algorithm, it is not necessary for an RTI to resolve the recursion nor any deadlock. For optimistic services, a simple mechanism without rollback in an RTI is also introduced; therefore, it can greatly simplify the development of an RTI.

1 INTRODUCTION

The High Level Architecture (HLA) has been developed to provide a common architecture for distributed modeling and simulation. In September 2000, HLA was adopted as the IEEE 1516 standard. The Runtime Infrastructure (RTI) is the software as an implementation of the HLA interface specification.

Time management is a great challenge for large-scale simulations in various RTIs, especially for those with a large amount of data communication. In those applications, an RTI may order a large amount of Time Stamp Order (TSO) messages. For example, suppose that a simulation has 1,024 federates with the same execution code. All federates call the timeAdvanceRequest (TAR) service to advance logical time step by step. For each step, everyone sends a TSO message subscribed by all other federates. Thus, the RTI in the simulation has to order 1,023*1,024 TSO messages. In an RTI with the same architecture as RTI1.3-NG (DMSO 2000), each federate has its own Local RTI Component (LRC). Therefore, 1,024 LRCs shall cooperate to order those 1,023*1,024 messages.

The HLA time management contains the conservative and optimistic mechanisms. In HLA, a federate can call the enableTimeRegulation service to determine if it regulates other federates' advancement, or call the enableTimeConstrained service to be constrained by other federates. In this paper, we call those services as policy-setting services. Greatest Available Logical Time (GALT) is the maximum logical time that a federate may advance securely. GALT is also called Lower Bound Time Stamp (LBTS) in HLA 1.3. The computation of GALT is a key problem for efficiently implementing HLA time management, and it is virtually different from those LBTS algorithms in traditional area of Parallel Discrete Event Simulation (PDES). For example, the important concepts of Local Virtual Time (LVT) and Global Virtual Time (GVT) in PDES have disappeared in the HLA time management.

By far, we have not yet found any RTI developers to publish the detail implementation of the HLA time management. This paper will introduce the implementation mechanisms of time management services in StarLink, which is an RTI with central architecture. StarLink only comprises a unique central RTI server, and the communication between a federate and the server is accomplished by underlying CORBA middleware (Liu, Wang and Yao 2004a). Two issues are very important when implementing time management, i.e. the algorithm of GALT computation and implementation of optimistic services. Complicated algorithms may result in an RTI's low efficiency and reduction of the RTI's application scales. However, such questions are resolved in StarLink easily and smartly. Although StarLink is an RTI with central architecture, its ideas may be general and can be reused into other RTIs with different architectures such as RTI1.3-NG, pRTI

(Pitch Technologies 2006), MÄK RTI (MÄK Technologies 2006), DRTI (FDK 2006), etc. In the next section, the GALT algorithm in StarLink is introduced. An important characteristic is that there is no recursion in the simple algorithm, and it doesn't resolve any deadlock caused by recursion. The implementation of policy-making services is described in the third section. Then the implementation algorithms of conservative time management services are presented. In the fifth section, the principles of optimistic time management services are introduced and especially the zero-saving mechanism is also discussed. The mechanism doesn't require an RTI to save or restore any data at all when a message is sent or retracted; therefore, this can greatly simplify the development of an RTI. We believe that those algorithms in the paper are rather useful for RTI developers to implement the HLA time management.

2 COMPUTING GALT

GALT expresses the greatest logical time to which an RTI can grant a federate's advance without having to wait for other joined federates. That is to say, GALT is the maximum logical time a federate can advance to. An RTI must compute each federate's GALT correctly and efficiently. The computation of GALT is the basic issue to implement time management services. It is rather complicated to compute GALT as stated in these papers (Fujimoto 2000; Kuhl, Weatherly and Dahmann 1999; Riley, Fujimoto and Ammar 2000). An RTI must consider each federate's time advance services, status, lookahead (Fujimoto 1988; Fujimoto 1997) and TSO messages, etc.

The message sent firstly must arrive firstly within StarLink, and the FIFO queuing mechanism is insured by underlying CORBA middleware. The following algorithm named stature-measuring is used to compute GALT. In the algorithm, a federate's stature is represented by the symbol S. A federate's stature is correlative with its time advancing status. In this paper, five states are defined for a federate, i.e. Grant, TAR, TARA, NMR and NMRA. The Grant state means a federate is not in time advancing state. The rest states mean a federate is in time advancing state by calling the timeAdvanceRequest, timeAdvanceRequestAvailable (TARA), nextMessageRequest (NMR) or nextMessageRequestAvailable (NMRA) service. However, a federate can also call the flushQueueRequest (FQR) service to advance logical time. As the FQR service is always granted by RTI, the service is not considered on computing GALT in StarLink though the computation for this service is nearly same as the NMRA service.

Definition 1 *A federate's stature S(i) is defined as*

$$S(i) = \begin{cases} T(i) + L(i); \ for\ Grant / TAR / TARA \\ \min\{T(i), LETS(i)\} + L(i); \ for\ NMR / NMRA. \end{cases}$$

Where *S(i)* means the stature of federate *i*. When federate *i* is in the Grant state, *T(i)* is its current logical time and *L(i)* means its actual lookahead; when federate *i* is in time advancing state, *T(i)* is the logical time to which the federate requests to advance and *L(i)* is the actual lookahead after its request is granted and its computation was presented in Carothers, Weatherly, Fujimoto, and Wilson (1997). *LETS(i)* is the least time stamp in federate *i*'s TSO queue.

Algorithm 1 *For any federate i, its GALT is determined by other federates' statures in the federation. GALT(i) is computed by*

$$GALT(i) = \min\{S(j)\}, i \neq j.$$

From the GALT algorithm, we know that only two federates with least statures are significant for computing GALT.

Definition 2 *The minimal federate is that with least stature in the federation.*

Definition 3 *The penultimate federate is that with least stature except the minimal federate in the federation.*

At any moment, only one minimal federate and one penultimate federate are designated in the federation. If there exist any other federates which have the same stature as the minimal federate or the penultimate federate, these federates are no longer called the minimal or the penultimate. In addition, the penultimate federate's stature may be equal to that of the minimal federate. From algorithm 1 and these definitions, we know that the following two theorems are correct apparently.

Theorem 1 *Anyone except the minimal federate has the GALT which is equal to the minimal federate's stature.*

Theorem 2 *The minimal federate's GALT is equal to the penultimate federate's stature.*

Any other federate's stature except the minimal federate may decrease because of TSO messages with less time stamp. Consider a simulation with federates *i* and *j*. Suppose that both federates are in time granted states, their logical time is *T*, their lookahead is 1, and their TSO message queues are empty. When federate *i* calls the NMR service to request to advance to *T+3*, the federate shall be suspended because *i*'s GALT is *T+1*, which is smaller than its request time *T+3*. Now we know that $S(i) = \min\{T+3, +\infty\}+1=T+4$ from definition 1. If federate *j* sends a TSO message with logical time *T+2* to federate *i*, $S(i) = \min\{T+3, T+2\}+1=T+3$. Thus, the stature of federate *i* decreases from *T+4* to *T+3*. However, it is provable that the penultimate federate's stature must not decrease to be less than the minimal federate's. In addition, the accurate stature of the penultimate federate is not important for the minimal federate's advance at all. The penultimate federate shall not hold back the advance of the minimal federate so far as the penultimate federate has larger stature. The GALT algorithm can ensure the minimal federate to advance its logical time correctly. Although there may exist

any federate whose stature decreases during the federation execution, the algorithm can compute the minimal federate's stature correctly so that it can compute any other federates' GALTs correctly according to theorem 1. Thus, the algorithm can ensure federates to advance their logical time correctly.

3 POLICY-SETTING SERVICES

According to the IEEE 1516 standard, a federate can use enableTimeRegulation, disableTimeRegulation, enableTimeConstrained and disableTimeConstrained to determine its time management modes. We call them policy-setting services in this paper. To send TSO messages, a federate shall call the enableTimeRegulation service to become a time regulating federate. A time regulating federate may call the disableTimeRegulation service to avoid sending TSO messages. To receive TSO messages, a federate shall call the enableTimeConstrained service to be a time constrained federate. A time constrained federate may call the disableTimeConstrained service to avoid receiving TSO messages. A time regulating federate influences the advance of a time constrained federate. This paper describes the algorithms of the enableTimeRegulation and enableTimeConstrained services. The other two services are relatively easier to implement. The enableTimeRegulation service has the following prototype.

```
void enableTimeRegulation (RTI::LogicalTime
theLookahead);
```

Algorithm 2 is the implementation of the service.

Algorithm 2 *enableTimeRegulation*

1. Process exceptions.
   (a) If the calling federate has not joined the federation execution, the RTI::FederateNotExecutionMember exception is thrown.
   (b) If the calling federate is a time regulating federate (The *bTimeRegulating* flag is equal to the enumeration constant ENABLED), the RTI::TimeRegulationAlreadyEnabled exception is thrown.
   (c) If the calling federate is in pending on calling the enableTimeRegulation service (The *bTimeRegulating* flag is equal to the enumeration constant PENDING), the RTI::RequestForTimeRegulationPending exception is thrown.
   (d) If the parameter *thelookahead* is less than zero, the RTI::InvalidLookahead exception is thrown.
   (e) If the calling federate is in time advancing state, the RTI::InTimeAdvancingState exception is thrown.
2. Compute logical time.

```
/* currentTime is a federate's actual logical
time, which is initialized to zero. */
maxTime = currentTime + theLookahead;
for(i = first constrained federate; all con-
strained federates; i is not the calling fed-
erate, i++) {
    if(federate i's logical time ≥ maxTime) {
        maxTime = federate i's logical time;
        if(federate i is in the Grant state
which is granted by calling the TAR/NMR ser-
vice at least once) {
            /* EPSILON can be set as the mini-
mal double number which a complier can iden-
tify. But a better way is to set a Boolean
flag rather than using such a constant. */
            maxTime = maxTime + EPSILON;
            // 0<EPSILON<<1
        }
    }
}
pendingTime = maxTime - theLookahead;
```

3. Set lookahead and pending flag.

```
lookahead = theLookahead;
bTimeRegulating = PENDING;
```

4. Send Receive Order (RO) messages.
   Send all RO messages to the calling federate.
5. Judge whether the calling federate is granted to be regulating.

```
if(the calling federate is time constrained){
    /* The invocation of this service shall be
considered an implicit TARA service invoca-
tion. */
    bTimeAdvanceRequestAvailable = PENDING;
    Compute the calling federate's GALT;
    if(pendingTime ≤ GALT) {
        All messages with time stamp ≤ pend-
ingTime in the calling federate's TSO queue
are sent to the federate;
        currentTime = pendingTime;
        bTimeRegulating = ENABLED;
      call timeRegulationEnabled(currentTime);
       bTimeAdvanceRequestAvailable = ENABLED;
    } else {
        All messages with time stamp ≤ GALT
are sent to the calling federate;
    }
}else{
    currentTime = pendingTime;
    bTimeRegulating = ENABLED;
    call timeRegulationEnabled(currentTime);
};
```

The enableTimeConstrained service has the following prototype.

```
void enableTimeConstrained (RTI::LogicalTime
theLookahead);
```

Algorithm 3 is the implementation of the service.

Algorithm 3 *enableTimeConstrained*

1. Process exceptions.
   (a) If the calling federate has not joined the federation execution, the RTI::FederateNotExecutionMember exception is thrown.
   (b) If the calling federate is time constrained (The *bTimeConstrained* flag is equal to the enumeration constant ENABLED), the RTI::TimeConstrainedAlreadyEnabled exception is thrown.
   (c) If the calling federate is in pending on the enableTimeConstrained service (The *bTimeConstrained* flag is equal to the enumeration constant PENDING), the RTI::RequestForTimeConstrainedPending exception is thrown.
   (d) If the calling federate is in time advancing state, the RTI::InTimeAdvancingState exception is thrown.
2. Set pending flag.

```
bTimeConstrained = PENDING;
```

3. Compute GALT.
4. Judge whether the calling federate is granted to be constrained.

```
if(the calling federate is time regulating){
    if(currentTime ≤ GALT) {
      bTimeConstrained = ENABLED;
     call timeConstrainedEnabled(currentTime);
    }else{
      /* In StarLink, a regulating but not
constrained federate can advance without be-
ing regulated by any other federates. If the
regulating federate goes farther from other
federates and now it wants to be a con-
strained federate by calling this service, we
suspend the federate and don't allow it to
call TAR/TARA/NMR/NMRA/FQR to advance again.
But we think the method is optional and it is
still reasonable if we allow the federate to
advance further. */
      bTimeAdvanceRequestAvailable = PENDING;
      pendingTime = currentTime;
    }
}else{
    if(GALT != +∞){ // GALT's original value
        if(currentTime ≤ GALT) {
            currentTime = GALT;
            bTimeConstrained = ENABLED;
            call timeConstrainedEnabled (cur-
rentTime);
        }
    }else{
        bTimeConstrained = ENABLED;
        call timeConstrainedEnabled (current-
Time);
    }
};
```

4 CONSERVATIVE ADVANCE SERVICES

The conservative mechanism in the HLA time management ensures that each federate can receive TSO messages in time stamp order, which guarantees the federation to be executed correctly. The conservative advance services in IEEE 1516 are TAR, TARA, NMR and NMRA. When a federate call any conservative services to advance logical time, an RTI shall call the timeAdvanceGrant (TAG) service to grant the federate's advance if the requested logical time is secure. Otherwise, the request shall be suspended and the federate shall be in time advancing state.

In addition, when a federate's request is granted, the RTI shall look up other federates to be suspended because of calling the TAR, TARA, NMR, NMRA, enableTimeRegulation and enableTimeConstrained services. The RTI shall recomputed these federates' GALTs and determine if they can resume and go on execution. This paper uses the pushFederates function to express this procedure. The algorithms of the TAR and NMR services are described in this section. The other two services TARA and NMRA have similar results.

The TAR service has the following prototype.

```
void timeAdvanceRequest (RTI::LogicalTime
theTime);
```

Algorithm 4 is the implementation of the service.

Algorithm 4 *timeAdvanceRequest*

1. Process exceptions.
   (a) If the calling federate has not joined the federation execution, the RTI::FederateNotExecutionMember exception is thrown.
   (b) If the parameter *theTime* is not correct, the RTI::InvalidLogicalTime exception is thrown.
   (c) If the calling federate is in time advancing state, the RTI::InTimeAdvancingState exception is thrown.
   (d) If the parameter *theTime* is less than the calling federate's current logical time, the RTI::LogicalTimeAlreadyPassed exception is thrown.
   (e) If the calling federate is in pending on calling the enableTimeRegulation service (The *bTimeRegulating* flag is equal to the enumeration constant PENDING), the RTI::RequestForTimeRegulationPending exception is thrown.
   (f) If the calling federate is in pending on calling the enableTimeConstrained service (The *bTimeConstrained* flag is equal to the enumeration constant PENDING), the

RTI::RequestForTimeConstrainedPending exception is thrown.

2. Set pending flag and pending time.

```
bTimeAdvanceRequest = PENDING;
pendingTime = theTime;
```

3. Send RO messages.
   Send all RO messages to the calling federate.
4. Judge whether the TAR request is granted.

```
if(the calling federate is time constrained){
    Compute the calling federate's GALT;
    if(theTime < GALT) {
        call   grantFederateAdvancing(theTime)
to grant the calling federate to theTime.
    }else{
        All messages with time stamp ≤ GALT
are sent to the calling federate;
    }
}else{
    call  grantFederateAdvancing(theTime)  to
grant the calling federate to theTime.
}
```

5. Call the pushFederates function to advance other pending federates.

In algorithms 4 and 5, the grantFederateAdvancing function mainly does following things.

1. Look up if the calling federate is in lookahead-pending state. When a federate calls the modify-Lookahead service to decrease its lookahead, the actual lookahead can not be decreased immediately and it must be decreased gradually during the federation. If the calling federate is in lookahead-pending state, the actual lookahead should be recomputed as stated in Carothers, Weatherly, Fujimoto, and Wilson (1997).
2. Send all TSO messages with time stamp less than or equal to the granted time to the calling federate.
3. Call the TAG service to grant the federate to advance its logical time. The *bTimeAdvanceRequest* or *bNextMessageRequest* flag is set to be ENABLED.

The NMR service has the following prototype.

```
void   nextMessageRequest   (RTI::LogicalTime
theTime);
```

Algorithm 5 does the implementation of the service.

Algorithm 5 *nextMessageRequest*

1. Process exceptions.
   This is the same as algorithm 4.
2. Set pending flag and pending time.

```
bNextMessageRequest = PENDING;
pendingTime = theTime;
```

3. Send RO messages.
   Send all RO messages to the calling federate.
4. Judge whether the NMR request is granted.

```
if(the calling federate is time constrained){
    Compute the calling federate's GALT;
    //LETS is +∞ for an empty TSO queue.
    LETS = min{TSO}; // the minimal time stamp
in the federate's TSO queue.
    if(LETS ≤ theTime && LETS < GALT) {
        call  grantFederateAdvancing(LETS)  to
grant the federate to LETS.
    }else if(theTime < GALT) {
        call   grantFederateAdvancing(theTime)
to grant the federate to theTime.
    }
}else{
    call  grantFederateAdvancing(theTime)  to
grant the federate to theTime.
}
```

5. Call pushFederates to advance other pending federates.

5 OPTIMISTIC ADVANCE SERVICES

The optimistic advance services in IEEE 1516 are FQR, retract and requestRetraction. The former two services are implemented within RTI, and the latter callback service is supplied by federate.

The optimistic mechanism in HLA is different from that in PDES. In PDES, there exists the typical domino phenomenon, i.e. an optimistic process may roll back its start. But a federate mustn't roll back its logical time and there doesn't exist the phenomenon during the HLA federation execution. As an HLA federate can only advance its logical time by calling the TAR, TARA, NMR, NMRA and FQR services, it cannot call any other services to roll back logical time. Otherwise, the federate shall receive the exception 'RTI::LogicalTimeAlreadyPassed' from its RTI. In addition, an RTI doesn't allow a federate to advance its logical time beyond GALT and the federate's logical time is always secure according to HLA standards; otherwise, the RTI shall not guarantee that a conservative federate can coexist with an optimistic federate. However, the received TSO messages for a federate may not secure. Some messages' time stamps may be larger than GALT. A federate may receive a message with larger time stamp during a request, and it may receive a message with smaller time stamp later. If the smaller message results in the confliction of states, the federate may roll back its states but not logical time, and it may retract its TSO messages sent by itself before. When its RTI receives the retract service, it calls the requestRetraction service to notify any federates that

have received the message to retract it. Thus, there are two important rules in the HLA optimistic advance mechanism.

Rule 1 *The logical time of a federate must not be rolled back.*

Rule 2 *The rollback occurs in a federate but not in an RTI.*

Here is another interesting phenomenon. A TSO message sent by a federate may be received by multiple federates. Some federates may receive it as a TSO message and some federates may receive it as an RO message. Moreover, a TSO message may be split into multiple messages and sent to one federate. Therefore, a federate may receive it not only as a TSO message but also an RO message in StarLink.

We think that an RTI should call the requestRetraction service to notify all receivers when a federate retracts a TSO message. If a federate receives more than one split message, the RTI only notifies the federate once. Of course, it is the federate's responsibility to determine if the requestRetraction service is responded.

The key issue in implementing optimistic services is to correctly notify all receivers to retract messages. This paper introduces the zero-saving mechanism that doesn't require an RTI to save anything, and more detailed discussion can be found in Liu, Wang and Yao (2004b). Of course, an RTI may save any states correlative with optimistic services, but this will make the development of the RTI much more complicated.

5.1 Complexity of Saving in RTI

Next we discuss two available methods for an RTI to save and restore information about message retraction.

1. To create a list for each federate in the RTI. An item of the list includes a TSO message handle and the message's receivers. When a federate retracts a message, the RTI can know which federates shall be notified. This method has nothing to do with the HLA declaration management.
2. To create two lists for each federate in the RTI. One list represents the publication and subscription relations of a federate, and an item of another list includes a TSO message handle and the index of publication and subscription relation in the first list. When a federate retracts a message, the RTI can also know which federates shall be notified by looking up and matching two lists. This method is correlative with the HLA declaration management and data distribution management.

However, an RTI shall do a lot of work for both methods to save and restore retraction information, and corresponding processing code will inevitably spread into many HLA services such as time management, federation management, ownership management, object management, data distribution management, and even declaration management. In addition, these approaches may result in a few hard nuts to crack. For example, a federate shall not be time regulating thus it can no longer retract messages if the disableTimeRegulation service is successfully called. But the federate is able to retract messages again after it recalls the enableTimeRegulation service successfully. Now the annoying question is however an RTI should maintain these lists when a federate calls both services. This is similar for the ownership transferring of object instance attributes. But these problems do not exist in the zero-saving mechanism.

5.2 Zero-Saving Mechanism

In StarLink, the message handle type RTI::MessageRetractionHandle is defined as follows.

```
struct MessageRetractionHandle {
    UniqueID          theSerialNumber;
    FederateHandle    sendingFederate;
    LogicalTime       theTime;
    FederateHandleSeq receivingFederates;
};
```

According to the IEEE 1516 standard, when federate *i* sends a TSO message, an RTI should return the federate a message handle with the RTI::MessageRetractionHandle type. In StarLink, the variable *theSerialNumber* in the message handle is a counter, which is used to distinguish between different messages; the variable *sendingFederate* means the sending federate; *theTime* is the time stamp of the TSO message. If the subscribing federate *j* with handle *h* receives the message as a TSO message, *h* is inserted into the variable *receivingFederates*. Suppose that the number of federates is less than 1,000,000 in the federation, *1000000+h* is inserted into the set if the subscribing federate *j* would receive the TSO message as an RO message. The assumption is reasonable because it is difficult for most RTIs to support more than 100 federates nowadays. An alternative method is to adopt two variables which represent the receiving federates for RO and TSO messages respectively.

The zero-saving mechanism in StarLink means that the RTI returns the sending federate a message handle with all received federates whenever the federate sends a TSO message. Thus, when the sending federate calls the retract service to withdraw a TSO message, the RTI will call the requestRetraction service to correctly notify all received federates which are saved in the variable *receivingFederates* of the TSO message handle.

The FQR service has the following prototype.

```
void    flushQueueRequest    (RTI::LogicalTime
theTime )
```

Algorithm 6 is the implementation of the service.
Algorithm 6 *flushQueueRequest*

1. Process exceptions.
   This is the same as algorithm 4.
2. Compute the calling federate's GALT.
3. Compute the calling federate's current logical time.

```
if(the calling federate is not time con-
strained){
    currentTime = theTime;
}else{
    currentTime = min{LETS, GALT, theTime};
}
```

4. Compute the calling federate's actual lookahead.
5. Send messages.
   Send all RO and TSO messages to the calling federate.
6. Grant the FQR request.
   Call the timeAdvanceGrant(*currentTime*) service to grant the federate to advance to *currentTime*.
7. Call pushFederates to advance other pending federates.

The retract service has the following prototype.

```
void              retract              (const
RTI::MessageRetractionHandle& theHandle)
```

Algorithm 7 is the implementation of the service.
Algorithm 7 *retract*

1. Process exceptions.
   (a) If the calling federate has not joined the federation execution, the RTI::FederateNotExecutionMember exception is thrown.
   (b) If the calling federate is not regulating, the RTI::TimeRegulationIsNotEnabled exception is thrown.
   (c) If *theHandle.SendingFederate* is not the calling federate, the RTI::InvalidRetractionHandle exception is thrown.
   (d) If the calling federate is in the Grant state and *theHandle.theTime* ≤ *(currentTime + lookahead)* or the calling federate is in time advancing state and *theHandle.theTime* ≤ *(pendingTime+lookahead)*, the RTI::MessageCanNoLongerBeRetracted exception is thrown.
2. Define a set variable Φ.

```
For    each    federate    h    in    theHan-
dle.receivingFederates, do the following pro-
cedure.
    if(h < 1000000) { // the federate receives
a TSO message
        if(the message is in federate h's TSO
queue) {
            remove this message from federate
h's TSO queue;
        }else{
            Φ = Φ + {h};
        }
    }else{ // the federate receives an RO mes-
sage
        h = h - 1000000;
        if(the message is in federate h's RO
queue) {
            remove this message from federate
h's RO queue;
        }else{
            Φ = Φ + {h};
        }
    }
```

3. Empty the *theHandle.receivingFederates* variable.
4. Send the requestRetraction(*theHandle*) service to all federates in Φ.

6 CONCLUSION

The principles and efficient implementation of the HLA time management have always been a hotspot for RTI developers. This paper systematically explains the principles, and describes the algorithms of most time management services. On computing Greatest Available Logical Time (GALT), an algorithm without recursion is introduced, and the deadlock caused by recursion shall not be considered. A federate may roll back its states when using optimistic services. It is very complicated for an RTI to save and restore the information correlating to the rollback because an RTI shall consider many other HLA management services besides time management. This paper describes the zero-saving algorithm, which can greatly simplify the implementation of an RTI. In general, we shall put the saving information for future retraction in an RTI. But according to the zero-saving mechanism, an RTI can return federates the retraction information and the RTI saves and restores nothing. Therefore, the zero-saving algorithm is essentially a mechanism which transfers the saving operation in an RTI to a federate.

ACKNOWLEDGMENTS

This work was supported in part by *the National Grand Fundamental Research 973 Program of China* under the grant of No. 2005CB321804 and *the National Natural Science Foundation of China* under the grant of No. 60373024.

REFERENCES

Carothers, C. D., R. M. Weatherly, R. M. Fujimoto, and A. L. Wilson. Design and implementation of HLA time management in the RTI version F.0. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradottir, D. H. Withers, and B. L. Nelson, 373-380. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers [online]. Available via <http://www.informs-sim.org/wsc97papers/0373.PDF> [accessed March 18, 2006].

DMSO. 2000. RTI 1.3-Next Generation Programmer's Guide Version 6 [online]. Available via <http://hla.dmso.mil> [accessed April 8, 2002].

FDK. 2006 [online]. Available via <http://www.cc.gatech.edu/computing/pads/papers.html> [accessed March 18, 2006].

Fujimoto, R. M. 1988. Lookahead in Parallel Discrete Event Simulation. In *1988 International Conference on Parallel Processing* 3: 34-41.

Fujimoto, R. M. 1996. HLA Time Management: Design Document. College of Computing Georgia Institute of Technology Atlanta [online]. Available via <http://www.cc.gatech.edu/computing/pads/papers.html> [accessed March 18, 2006].

Fujimoto, R. M. 1997. Zero Lookahead and Repeatability in the High Level Architecture. In *1997 Spring Simulation Interoperability Workshop* [online]. Available via <http://www.cc.gatech.edu/computing/pads/papers.html> [accessed March 18, 2006].

Fujimoto, R. M. 2000. *Parallel and distributed simulation systems*. New York: John Wiley & Sons.

Kuhl, F., R. Weatherly, and J. Dahmann. 1999. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, Upper Saddle River, NJ.

Liu, B. Q., H. M. Wang, and Y. P. Yao. 2004a. Key techniques of a hierarchical simulation runtime infrastructure-StarLink. *Journal of Software* 14(1): 9-16 [online]. Available via <http://www.jos.org.cn/paper/detail.asp?id=1765> [accessed June 10, 2006].

Liu, B. Q., H. M. Wang, and Y. P. Yao. 2004b. Implementation of optimistic advancing mechanism in RTI. *Journal of Software* 14(3): 338-347 [online]. Available via <http://www.jos.org.cn/paper/detail.asp?id=1803> [accessed June 10, 2006].

MÄK Technologies. 2006 [online]. Available via <http://www.mak.com/rti.htm> [accessed March 18, 2006].

Pitch Technologies. 2006 [online]. Available via <http://www.pitch.se/prti> [accessed March 18, 2006].

Riley, G. F., R. Fujimoto, and M. H. Ammar. 2000. Network aware time management and event distribution, College of Computing Georgia Institute of Technology Atlanta [online]. Available via <http://www.cc.gatech.edu/computing/pads/papers.html> [accessed March 18, 2006].

AUTHOR BIOGRAPHIES

BUQUAN LIU received his B.S. degree in computer science from Nanjing University in 1991. His M.S. and Ph.D. degrees were received in School of Computer from National University of Defense Technology (NUDT) in 1998 and 2004 respectively. He has achieved 2 Provincial Science and Technology Advance Awards and 1 patent of *the design of hierarchical RTI servers based on interoperability protocol*. Now he is an associate professor of the school and his interests are distributed simulation and high performance computing. His e-mail address is <bqliu@nudt.edu.cn>.

YIPING YAO is a professor of School of Computer in National University of Defense Technology. In this school, he received his M.S. and Ph.D. degrees in 1987 and 2004 respectively. He received his B.S. degree in computer science from Huazhong University of Science and Technology in 1985. At present, he has achieved 2 second-class National Science and Technology Advance Awards and 8 Provincial Science and Technology Advance Awards. More than 40 papers and 3 monographs were also published. His research areas are distributed simulation and virtual reality. His e-mail address is <ypyao@nudt.edu.cn>.

JING TAO is an associate professor in the School of Computer at the National University of Defense Technology. She received her B.S. and M.S. degrees in the school in 1992 and 1998 respectively. She has won 3 Provincial Science and Technology Advance Awards. Her interests are also distributed simulation and high performance computing. Her e-mail address is <ellen5702@tom.com>.

HUAIMIN WANG is a professor in the School of Computer at the National University of Defense Technology. He received his Ph.D. degree in computer science in 1992. He is a member of the Editorial Board of *Chinese Journal of Computers* and *Journal of Computer Science and Technology*. Dr. Wang has served as a member of the Expert Committee for *Computer Software and Hardware of the National High Technology Research and Development Program of China* (863 Program). In 2003, he was awarded one 2nd class National Science and Technology

Advance Award. Up to now, he has published more than 90 papers and directed 20 graduate students. His research focuses on distributed object, agent technology, grid computing and network security. His e-mail address is <whm_w@163.com>.