FORMAL SPECIFICATION OF CHECKPOINTING ALGORITHMS

Gavril Godza, Valentin Cristea
Radu Mateescu

Computer Science Department,
POLITEHNICA University of Bucharest
{gavrilg, valentin}@cs.pub.ro
INRIA Rhones-Alpes, Grenoble, France
Radu.Mateescu@inria.fr

Abstract: To attain good reliability of any software product, its development should begin with a *formal specification*. This way, it is very easy to perform correctness verification and some performance evaluation information is also available. Formal specification of distributed systems is frequently used for a cost-effective error detection and correction during the initial phase of the software development process. In order to achieve this goal, several tools have been developed for program analysis, code generation, simulation, testing, test data generation. It is important to notice that this kind of tools and models can be used for general distributed system applications.

Keywords: distributed system, fault tolerance, rollback recovery, checkpointing algorithm, formal specification, LOTOS, CADP.

1. INTRODUCTION

There are several types of failures that can occur in a distributed system: software crashes, disk failures, and bus errors, network errors, processor errors and power losses. The first consequence of such a failure is a *data loss*, which can lead to an inconsistent *system state*. The state of a distributed system depends on the states of each process in the system. But due to inter-process communication the states of processes depend on one another. If a component of a *distributed* system fails, its failure could propagate to and affect other components. Especially for these systems *reliability* is a very important issue: it would be very useful to have the system working even in the presence of some failures.

1.1. Formal specifications

Formal program verification is the most reliable technique because it is complete, but it is also expensive. There are several formalisms proposed to describe parallel and distributed systems.

- Model Based Approach that uses mathematics, logic and set theory (Z and VDM languages).
- Algebraic Approach.
- Concurrency Approach - CSP, CCS, LOTOS, ESTELLE, SDL.

In the ‘80s several hybrid models were proposed, but ISO or ITU-T (former CCITT) has adopted three of them, called Formal Description Techniques, as standards. These specification languages are **ESTELLE** (Extended Finite State Machine Language), **SDL** (Specification and Description Language) and **LOTOS** (Language Of Temporal Ordering Specification). These languages have some common features. They allow a hierarchical structure of models to be specified, where subsystems are running in parallel and are communicating through message exchanges. Each specification contains a **control part** (how does the system react to events?) and a **data part** (what kinds of information exist and how does a message look like?).

ESTELLE is a FDT defined within ISO for specification of distributed, concurrent information processing systems (communication protocols and services). An ESTELLE system consists of several module instances (tasks), with asynchronous behavior. Each module has a number of input/output access points that are connected by bi-directional links. Messages (interactions) are stored in a FIFO queue in each interaction point allowing for a non-blocking message sending. Another way to communicate is by the way of common variables (restricted to the son - parent communication). Two kinds of task parallelism can be expressed in ESTELLE: asynchronous parallelism and synchronous parallelism.

SDL is not directed specifically to describing telecommunications services, but it is a general-purpose description language for communication systems. The basis for description of behavior description is the communicating Extended State Machines that are represented by processes. Communication is represented by signals and can take place between processes or between processes and the environment of the system model. A process has an infinite FIFO input queue, where incoming signals are placed. A process is either waiting in a state or performing a transition between two states. A transition is initiated by a signal in the input queue, from which it is removed. During a transition variables can be manipulated, decisions made, new process instances created, signal sent, etc. Processes can be created statically or dynamically.

LOTOS is a FDT standardized by ISO for the design of distributed systems, and in particular for OSI services and protocols. Experts of the ISO FDT group developed LOTOS from 1981 to 1988; it has now the status of International Standard [IS8807]. Unlike FDTs based on the state representation of a system, LOTOS describes a system by defining the temporal relations between externally observable events at so-called event gates. LOTOS is composed of two parts: a **process algebraic** part based on Milner's Calculus of Communicating Systems (CCS) and on Hoare's Communicating Sequential Processes (CSP), and a data algebraic part based on the abstract data type language (ACT ONE). These two aspects of LOTOS are complementary and independent: the process algebra is used to model dynamic behaviors of systems, and ACT ONE is used to model data structures and value expressions, which allows handling data without unnecessary implementation details.

1.2. Fault Tolerance

A *system failure* is a behavior that doesn't follow the system specification (which should be complete, correct and consistent). After a system failure occurs, the service provided by the system is no longer that stated in its specification. A computing system is fault tolerant if it produces correct results, even if some components are faulty. The basic way to fight against failures has been proposed by Von Neumann: the use of *protective redundancy*. There are some extra components (hardware, software or time components), useless for a normal behavior, but which mask failures after they appear. Each calculation step is added some extra operations, performed by different software or hardware entities. So fault tolerance can be obtained in different ways: by adding extra processors, memory or communication links (hardware redundancy), by adding extra software modules to deal with extra hardware (software redundancy), by using extra time to implement several methods (time redundancy).

A *distributed system* consists of a set of autonomous loosely coupled nodes, interconnected through a communication network. Communication is made using a message passing approach, because in general there is no shared memory. From a fault tolerance point of view, a very important issue is the absence of a global clock. Each node has some atomic components that can fail (a processor, a local clock, a network interface, a stable storage and some software modules). Links between nodes can also fail, but this kind of failure could be viewed as processor failures. The way in which nodes are linked - the system *topology* - is another critical component: the better the connectivity is, the higher the reliability is. A distributed application consists of several concurrent processes (or threads) that compete or cooperate to provide a service. These processes communicate using *communication channels* that are logical links. Distributed systems considered in practice are said to be *synchronous*: each action has a finite duration. This way one can figure out if a delay is caused by a processor crash or it is a communication delay.

Error recovery involves restoring an error-free state from an erroneous one. Error recovery schemes are usually classified as:

- Forward error recovery, when the state that has just been found to be in error is further used, in an attempt to generate the correct state, using some error correction or error compensation techniques.
- Backward error recovery, when some processes are rolled back to a previous error free state, and then the computation is restarted.

Backward error recovery makes possible to recover from an arbitrary fault, thus it can be considered as a general recovery mechanism. It is based on some *recovery points*, that allow saving and restoring the state of a process. These recovery points can be obtained using several techniques, one of the most used being *checkpointing*, in which a set of local checkpoints (saved local states) is determined during normal computation, such us upon a failure occurrence, a rolled back computation can be resumed from this set. In order to achieve this, the checkpoints should be *consistent*. A consistent global checkpoint is a set of local checkpoints, one from each process, if there are no causal dependencies between them.

In a consistent checkpointing algorithm may appear a stable storage contention between processes, because they try to save the state on about the same time. To avoid this arbitrarily staggered consistent checkpoints could be used.

Checkpointing algorithms are classified as

- *Synchronous* (*coordinated*), where processes synchronize their checkpointing activities, so that a globally consistent set of checkpoints is always maintained. This involves message overhead and performance degradation because process execution may be suspended during coordination. Coordinated checkpointing periodically saves the state on stable storage.
- *Asynchronous* (*uncoordinated*), when processes do not coordinate with others when taking a checkpoint. This provides maximum process autonomy, but some checkpoints may be inconsistent. So it is worth to reduce the number of useless checkpoints. One method for doing this is a communication-induced checkpoint, where some checkpointing activity is triggered by the message pattern and knowledge gained about dependencies. Communication induced checkpointing uses a lazy coordination, by piggybacking control information on application messages. There are *basic* checkpoints and *forced* checkpoints and two distinct categories: *model-based* (that assures a deterministic behavior and domino-free property) and *index-based* (that assigns sequence numbers to local checkpoints and enforces consistency for the same sequence number).

Checkpointing algorithms can use a periodic state saving, an incremental state saving or a hybrid approach. Another possibility is diskless checkpointing, used for long running computations, without relying on a stable storage. The stable storage is replaced with memory and processor redundancy, eliminating the main source of overhead (but not for free). Recovery uses replacement processors that calculate the state of faulty processors.

There is another classification of checkpointing algorithms:

- Blocking algorithms, that force all relevant processes to block their computation during checkpointing.
- Non-blocking algorithms, that use a sequence number scheme, can be centralized or distributed.
- Combined approach, which forces only a minimum number of processes to take checkpoints. It involves system messages and not computation messages.

The notion of consistent global checkpoint is fundamental to many areas of distributed systems:

- Parallel and distributed debugging
- Distributed computing
- Fault tolerance
- Detection of stable properties
- Migrating processes between processors

2. THE FORMAL DESCRIPTION TECHNIQUE LOTOS

In the past years the standardized FDT LOTOS has received a considerable attention from the research community. **LOTOS** is a Formal Description Technique (FDT) standardized by ISO for the design of distributed systems, and in particular for OSI services and protocols. Experts of the ISO FDT group developed LOTOS from 1981 to 1988; it has now the status of International Standard [IS8807]. Unlike FDTs based on the state representation of a system, LOTOS describes a system by defining the temporal relations between externally observable events at so-called event gates. LOTOS is composed of two parts: a **process** algebraic part based on Milner's Calculus of Communicating Systems (CCS) and on Hoare's Communicating Sequential Processes (CSP), and a **data** algebraic part based on the abstract data type language (ACT ONE). These two aspects of LOTOS are complementary and independent: the process algebra is used to model dynamic behaviors of systems, and ACT ONE is used to model data structures and value expressions, which allows handling data without unnecessary implementation details.

LOTOS has been widely used for the specification of large data communication systems. It is mathematically well defined and expressive: it allows the description of concurrency, non-determinism, and asynchronous communications. It supports various levels of abstraction and provides several specification styles. In LOTOS a system is viewed as a hierarchy of (parallel) processes that can interact with each other or with the environment. In its early ages have no temporal constraints. A process is described using behavior expressions. Communication and synchronization are done through rendezvous, without shared memory. Good tools exist to support specification, verification and code generation. LOTOS is one of the few process algebras that have moved out of the theoretical community.

LOTOS is currently under revision in ISO under the ELOTOS (Enhancements to LOTOS) activity to improve the data type language and add features such as a module system and quantitative time.

Verification of the desired properties of a specification may be divided into several categories:

- Proof-theoretic (or axiomatic): where specifications are written in or translated into the notation of a proof system in which theorems may be proved using, for example, equational reasoning and term rewriting. Properties of systems specified in process algebra have been proved using compositional proof methods working with a given notion of equivalence: a system is broken into components which are shown to have certain properties that are together strong enough to imply the desired properties of the overall system.
- Model-checking (state/process based): A transition system is "captured" in some way by a machine representation (an automaton or some other finite state machine), which is generated by recursively applying the rules for transitional semantics. An algorithm, called a model checker, then can establish automatically and exhaustively whether or not desired properties hold for this representation (and hence for the transition system).
- Testing: is dependent upon how the system's behavior may be observed in its external interaction. Tests may be derived from an initial specification, and the resulting interaction with the implementation under test (IUT) simulated. This offers quickly some initial indications of whether an implementation satisfies certain requirements. The types of tests possible depend upon the language used.

The **CADP toolbox** is dedicated to the design and verification of communication protocols and distributed systems. It was initiated in 1986 at INRIA Rhones-Alpes, Grenoble, France and now the **CADP 99 beta t** version is available. It consists of several tools, that can be used in a command line style or through a graphic interface (EUCALYPTUS). The main tools are:

- CAESAR, CAESAR.ADT are compilers able to translate a LOTOS program into a finite state graph describing its exhaustive behavior.
- ALDEBARAN is a verification tool able to either compare or to minimize graphs with respect to bisimulation relations.
- TERMINATOR, EXHIBITOR, XSIMULATOR, EVALUATOR are tools that operate on the fly, providing respectively partial deadlock detection, incorrect execution sequence exhibition, interactive simulation and evaluation of temporal logic formulas.

3. BASIC CHECKPOINTING ALGORITHMS

There are several known checkpointing algorithms. All of these induce a communication and computation overhead that can lead to system performance degradation. This is why a checkpointing algorithm should be as simple as possible. We have chosen to analyze here some checkpointing algorithms that have (at least at the first sight) this property. The Sync-and-Stop (SNS) Algorithm is a representative for the blocking algorithm class, while the Chandy-Lamport (CL) algorithm represents the non-blocking class of algorithms.

1.3. The Sync-and-Stop (SNS) Algorithm

It is a very simple consistent checkpointing algorithm that basically shuts down the application to define a consistent cut and take a global checkpoint. There is a special, *coordinating* processor (P_c) that has the role to start and stop checkpoints. When it is time to start a checkpoint (on a periodical basis) the coordinator first stops the application and then broadcasts a special marker message to all other processors. When a regular processor P receives the marker message it stops running its program and waits for all sent application messages to be received. After that it sends the marker message back to P_c. After receiving the marker from all processes, P_c rebroadcasts it and takes its local checkpoint. After receiving the second marker, each regular process takes its local checkpoint and resends the marker to P_c. When P_c receives the message from each process, the checkpoint is complete. It has to rebroadcast the marker to let other processes know that the checkpoint is done. Before restarting the application, each process could perform some garbage-collection activity by removing old checkpoints.

1.4. The Chandi-Lamport (CL) Algorithm

It is more complex than the SNS algorithm and its main feature is that the application is not stopped: it interferes with the checkpointing algorithm. It also has a coordinating processor and it takes into account the communication links between processors. P_c starts the checkpoint by broadcasting a special message to all its neighbors. When a processor receives the special message and has not taken its local checkpoint yet, it broadcasts the special message to all its neighbors and right after that it takes its local checkpoint. Afterwards, if the same process receives an application message on a channel on which it has not received the special message yet, it must log the message because it is a cross-cut message. When there are no more cross-cut messages (all processors have received the special message on all their incoming channels) the local checkpoint is finished. Each process now notifies P_c by sending an acknowledgement. When P_c receives all acknowledgements it rebroadcast the message and when this last message is received the checkpoint is done and the garbage-collection operation could be performed.

4. FORMAL SPECIFICATION OF CHECKPOINTING ALGORITHMS

To obtain the formal specification for the checkpointing algorithms we considered that several interconnected processes cooperates by *message passing* for a long running finite, distributed computation The processes can access a stable storage, on which the checkpoints and message logs are stored. The system architecture is depicted in **Figure 1**, where each box stands for a LOTOS process.

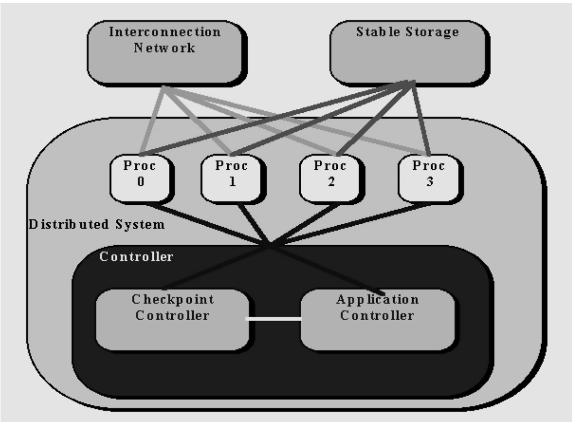

Figure 1. The system architecture.

The **Network** process models the interconnection network; it is in fact a buffer with a controllable capacity. The network topology resides in the data part, several topologies have been implemented: star, ring, bi-directional ring, and full connectivity. Messages are identified by a sender ID, a receiver ID, a tag and a body. A message can be send only if the network is not full. The **Storage** process models the stable storage. The **Proc** process is the actual application process, the core of the specification. The **Controller** process is the decision center; it decides when a message has to be sent and when a checkpoint is to be started; it has two distinct components, one that controls the distributed application and one that controls the checkpointing algorithm. The **Proc** and **Controller** processes model the distributed application. A process can send / receive an application message, perform an internal computation, participate to the checkpoint algorithm, fail and detect that another process has failed. Each process terminates after a finite number of exchanged messages. In the SNS algorithm, because of the sharp separation between application and checkpointing, we used a distinct LOTOS process for the checkpointing part, while in the CL case there is a single LOTOS process. Each of these components is implemented as some finite state machines.

The way in which the broadcast is performed is essential for an efficient checkpointing algorithm, so we have used broadcast algorithms that are constructed taking into account the network topology. This way we reduced the number of exchanged messages and simplified the specification. If a failure does occur, the recovery mechanism should be activated using a timeout technique. Because there is no explicit time in actual LOTOS, the timeout is modeled using a rendezvous. Another available one replaces the affected process. The state of the new process is read from the stable storage and then the computation is resumed. If a process fails, the others have to roll back to the previous recorded state, before resuming the computation.

5. RESULTS

First the ideal distributed system / application has been implemented, without any errors. Then the checkpointing algorithm has been added. Finally, the failure possibility and the recovery mechanism have been introduced. This gradually specification growth has lead from a system with tens of states to a system with millions of states, hardly to analyze. This is why the direct approach has been used only for simple systems, the complex ones being analyzed using a compositional approach (each system component has been individually minimized, and then all these intermediate results have been gathered).

The main analyzed cases are depicted in Table 1 and some results are presented in Table 2. However, more than 140 intermediate specifications have been analyzed. From these we can say that the ring version and the SNS algorithm have fewer states.

Table 1. Cases under study.

Ring	Star
Distributed system, no faults	Distributed system, no faults
Distributed system, SNS	Distributed system, SNS
Distributed system, CL	Distributed system, CL
Distributed system, SNS, faults, recovery	Distributed system, SNS, faults, recovery
Distributed system, CL, faults, recovery	Distributed system, CL, faults, recovery

The use of LOTOS disabling operator '**[>**' operator is prohibitive, because it leads to a state explosions, that can hardly be controlled. The smallest resulted graph (for ring topology, without faults) has 2776 states and

12639 transitions (527/1428 after minimization). The biggest resulted graph (for star topology, with checkpoint) has 166761 states and 17100248 transitions.

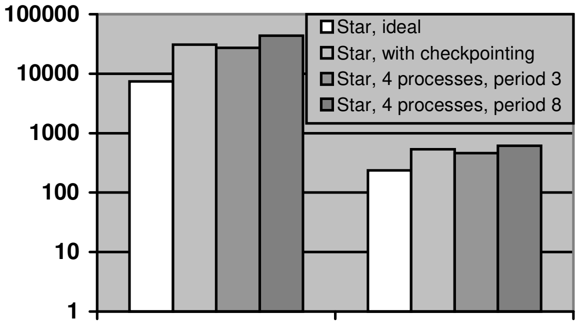

Figure 2. Number of states for star topology.

Table 2. Some results.

Case	States	Trans	BF
Initial			
Ring, ideal	2776	12639	4.55
Ring, with checkpointing	13368	46927	3.51
Star, ideal	7449	16960	2.27
Star, with checkpointing	30779	77334	2.56
Fully connected, ideal	588661	2857854	4.80
Ring, 4 processes, period 3	21144	61057	2.89
Star, 4 processes, period 3	27104	71859	2.65
Star, 4 processes, period 8	43670	182718	4.18
Star, 8 processes, period 3	500000	2500000	5.00
Minimised			
Ring, ideal	527	1428	2.71
Ring, with checkpointing	732	1418	1.94
Star, ideal	238	433	1.82
Star, with checkpointing	532	899	1.69
Ring, 4 processes, period 3	301	498	1.65
Star, 4 processes, period 3	458	808	1.76
Star, 4 processes, period 8	613	1018	1.66

The following properties has been verified using the CADP toolbox:

- All graphs are deadlock free.
- All graphs are livelock free.
- Each process terminates.
- Each SEND operation has a corresponding RECEIVE.
- There is no RECEIVE operation without a corresponding SEND (the corresponding predicate is shown below).

```
[(( not "SEND !0 !1 !SNDRCV !APP")* .
   "RECEIVE !1 !0 !SNDRCV !APP") |
   (( not "SEND !1 !2 !SNDRCV !APP")* .
     "RECEIVE !2 !1 !SNDRCV !APP") |
     (( not "SEND !2 !3 !SNDRCV !APP")* .
       "RECEIVE !3 !2 !SNDRCV !APP") |
       (( not "SEND !3 !0 !SNDRCV !APP")* .
         "RECEIVE !0 !3 !SNDRCV !APP")] false
```

Figure 3. The predicate that check that "There is no RECEIVE without a corresponding SEND".

- Safety and liveness of the checkpointing.
- Cut consistency.
- After a fault each process eventually terminates.

6. CONCLUSIONS AND FUTURE WORK

We can say that CADP is a very useful toolbox, comparing with other similar tools. It has several components that allow the formal specification and verification of distributed systems. It is available for SOLARIS, LINUX and Windows platforms, its portability being a great advantage. However, it requires many resources (especially memory). This is not a very big problem, because there is the compositional approach and memory becomes cheaper.

Formal specification is a promising issue for checkpointing, the resulting software being more reliable. However modeling faults and using dynamic data determine a state explosion, but this is a reasonable price paid.

This work will be continued by further refining the LOTOS specifications and by focusing on other interesting checkpointing algorithms and finally will lead to a real implementation.

7. BIBLIOGRAPHY

Baldoni, R., J. M. Helary, A. Mostefaoui, M. Raynal (1995). *Consistent Checkpointing in Message Passing Distributed Systems*. INRIA Rhone-Alpes, Rapport de Recherche No. 2564.

Baldoni, R., J. M. Helary, A. Mostefaoui, M. Raynal (1995). *On Modeling Consistent Checkpoins and the Domino Effect in Distributed Systems*. INRIA Rhone-Alpes, Rapport de Recherche No. 2569.

Bolognesi, T. , E. Briskma (1988). *Introduction to the ISO Specification Language LOTOS*.

Cao, G., M. Singhal (1998). *On Coordinated Checkpointing in Distributed Systems*. IEEE

Transaction on Parallel and Distributed Systems, vol. 9, No. 12, pp. 1213-1225.

Cristea, V., G. Godza (1997). *Formal Specification of Communicating Distributed Processes*, CSCS11 Conference, Bucharest, vol. II, pp.1-6.

Cristea, V., G. Godza (1999). *Developing Distributed Applications in Heterogeneous Environments*. A comparative study. CSCS12 Conference, Bucharest, vol. II.

Cristea, V., G. Godza (1999). *Some Synchronization and Performance Aspects in Message Passing Based Application Development*, CSCS12 Conference, Bucharest, vol. II.

Elnozahy, E. N., W. Zwaenepoel (1992). *An Integrated Approach to Fault Tolerance*. Rice University, 1992.

Elnozahy. E. N., (1993). *Manetho: Fault Tolerance in Distributed Systems Using Rollback Recovery and Process Replication*. Ph.D. Thesis, Rice University.

Elnozahy, E. N., W. Zwaenepoel (1994). *On the Use and Implementation of Message Logging*. Carnegie Mellon University.

Elnozahy, E. N., D. B. Johnson, Y. M. Wang (1996). *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. Carnegie Mellon University.

Garavel, H. (1999). *Description et Applications du Langage LOTOS*. ENSIMAG, INPG.

Garavel, H., L. Mounier (1996). *Specification and Verification of Various Distributed Leader Election Algorithms for Unidirectional Ring Networks*. INRIA Rhone-Alpes, Rapport de Recherche No. 2986.

Godza, G., (1998). *Fault Tolerance in Distributed Systems*, Technical Report, Bucharest.

Godza, G., (1999). *Fault Tolerant Distributed Systems*, The 3rd Sympozion "The Information Society", Bucharest.

Godza, G., (1999). *Evaluating Fault Tolerant Mechanism in Distributed Systems*, Technical Report, Bucharest.

Haj-Hussein, M., L. Logrippo (1996). *Specifying Distributed Algorithms in LOTOS*. University of Ottawa.

James, P., M. Endler, M. C. Claudel (1999). *Development of an Atomic-Broadcast Protocol Using LOTOS*. Universidade de Sao Paolo.

Logrippo, L., M. Faci, M. Haj-Hussein (1992). *An Introduction LOTOS: Learning by Examples*. Computer Networks and ISDN Systems vol 23, pp. 325-342.

Manivannan, D., M. Singhal (1999). *Quasi-Sunchronous Checkpointing: Models, Characterization and Clasification*. IEEE Transaction on Parallel and Distributed Systems, vol. 10, No. 7, pp. 703-713.

Manivannan, D., R. H. B. Netzer, M. Singhal (1997). *Finding Consistent Global Checkpointers in a Distributed Computation*. IEEE Transaction on Parallel and Distributed Systems, vol. 8, No. 6.

Planck, J. S. (1993). *Efficient Checkpointing on MIMD Architectures.*. Ph.D. Thesis, University of Tennessee.

Planck, J. S., K. Li, M. A. Puening (1998). *Diskless Checkpointing*. IEEE Transaction on Parallel and Distributed Systems, vol. 9, No. 10, pp. 972-986.

Stepien, B., L. Logrippo (1996). *Status-Oriented Telephone Service Specification: An Exercise in LOTOS Style*. University of Ottawa.

Turner K. J. (1989). *The Formal Specification Language LOTOS: A Course for Users*. University of Stirling, UK.

Vaidya N. H., (1999). *Staggered Consistent Checkpointing*. IEEE Transaction on Parallel and Distributed Systems, vol. 10, No. 7, pp. 694-702