# Units of Computation in Fault-Tolerant Distributed Systems*

Mohan Ahuja & Shivakant Mishra
Department of Computer Science & Engineering
University of California San Diego
La Jolla, CA 92093–0114, USA

## Abstract

*We develop a framework that helps in developing understanding of a fault-tolerant distributed system and so helps in designing such systems. We define a unit of computation in such systems, referred to as a molecule, that has a well defined interface with other molecules, i.e. has minimal dependence on other molecules. The smallest such unit—an indivisible molecule—is termed as an atom. We show that any execution of a fault-tolerant distributed computation can be seen as an execution of molecules/atoms in a partial order, and such a view provides insights into understanding the computation, particularly for a fault-tolerant system where it is important to guarantee that a unit of computation is either completely executed or not at all and system designers need to reason about the states after execution of such units. We prove different properties satisfied by molecules and atoms, and present algorithms to detect atoms in an ongoing computation and to force the completion of a molecule. We illustrate the uses of the developed work in application areas such as debugging, checkpointing, and reasoning about stable properties.*

## 1 Introduction

Reasoning about modern computing systems is difficult as, among other things, processes are distributed and may *fail* [1] at any time. However, with increasing use of these systems in every day life, it is of vital importance to understand such systems and reason about them correctly, so as to be able to design them in a better way. In this paper, we develop a framework that helps in understanding and so in designing such systems.

Similar work has been done in the past for systems in which process *failures* do not occur and in which the only *cause* and *effect* relationship is a *send* and the corresponding *receive* event [2]. We generalize these results. As far as possible, we present results for the most generalized situation, simply in terms of pairs of cause and effect events; these results can be used to develop details for systems with any specific cause and effect event pairs. We develop details for systems in which process failures may occur and messages may be multicasted; this involves giving due consideration to two new pairs of events, a *fail* event (event representing failure of a process) and the corresponding *delete* events (event of a process taking note of a fail event), and a *begin* event (event representing beginning of the execution of events on a process) and the corresponding *add* events (event of a process taking note of a begin event), and permitting multiple receive events corresponding to a send event.

We first extend the definition of a global snapshot [5] to our system model. A global snapshot in the absence of process failures is defined as a global state that contains send events corresponding to every receive event in the state. When process failures do occur, this definition is extended such that the global state contains fail (begin) events corresponding to a delete (add) event in the state.

Then we define *waves* and *wavefronts* and use them to define units of computations such that the execution can be divided into such units and each unit can be examined in isolation; it is easier to analyze an execution in smaller logical units than to analyze it as a whole. These units are termed as *molecules* and are defined by recognizing a direct causal relationship between different events in the system, i.e. which event in the computation causes another event to occur, and ensuring that a molecule contains either both cause and effect or neither. In light of the work done in atomic actions [11], importance of molecules becomes obvious. Definition of a molecule is essentially generalization and formalization (of this generalization) of an atomic action; an atomic action, for its system model, either executes completely or not at all as seen by the rest of the computation, affects the rest of the computation only by determining the states of processes in the atomic action at the end of the atomic action execution, and the rest of the computation affects the atomic action only by determining the states of these processes at the beginning of the

*This work is supported in part by NSF grants MIP-9111045 & MIP-9296204, and grants from Powell Foundation and Sun Microsystems

[1] Definitions of this and other italicized terms will be given in the paper.

atomic action execution. Generalization is to include any pair of cause and effect events. Smaller a molecule, easier it will be to analyze it, and smaller will be the work lost as a result of a failure during the execution of the molecule. So, understanding a system in terms of smallest molecules has advantages. We identify the smallest molecule, and term it as *atom*. We prove various properties satisfied by molecules and atoms. We show that molecules (so also atoms) can be partially ordered, and any *fault-tolerant* distributed computation is an execution of these molecules in that partial order.

We present outlines of algorithms to recognize an atom in an ongoing execution of such a computation, and to force completion of an ongoing molecule. Both of these algorithms are useful in different circumstances; for example, detecting atoms is useful for writing debuggers, while both detecting atoms and forcing completion of molecules are useful for checkpointing process state.

The framework developed results in a number of insights. Analyzing an execution of a fault-tolerant distributed computation in terms of (these smaller) molecules indeed simplifies understanding of the computation; we show so for two different computation models, i.e. communicating sequential processes and process group. For each of these computation models, we analyze the structure of the atoms, and the relations between them. For process group computation model, we also investigate the effect of specific properties of *membership protocols* on the structure of the molecules and atoms.

The rest of the paper is organized as follows. In Section 2, we describe the system model we assume, identify different types of events that are relevant to our work, and based on the state change resulting from their execution we describe cause and effect relationship between them. In Section 3, we give a definition of a global snapshot in *fault-tolerant* distributed system. In Section 4, we define waves, wavefronts, ground states, molecules, and atoms; and give their properties. In Section 5, we give outline of algorithms to detect atoms in an ongoing computation, and to force completion of an ongoing molecule. In Section 6, we show that viewing a fault-tolerant distributed computation in terms of atoms helps in understanding the computation. Finally, in Section 7, we describe some applications of this research, particularly important for fault tolerance. Proofs of all the theorems are given in [3].

## 2 System Model

We assume a distributed system that is asynchronous, comprising of a finite set $\mathcal{D}$ of processes represented by $p$, $q$, or $r$. $p$ communicates with other processes only by sending/multicasting/broadcasting, (referred to as sending) and receiving messages. Communication is assumed to be reliable, so no messages are lost in transit and no spurious messages are added. Communication is also assumed to be asynchronous, so these messages are received in a finite but unbounded time. For simplification, we assume that for any two processes $p$ and $q$ there is a uni-directional channel $c_{p,q}$ along which $p$ sends messages to $q$; unless otherwise specified we do not assume that messages are received in any specific order.

We assume that only processor or process failures occur; a processor fails by *crashing*, i.e. silently without making any incorrect state transition[7]; a process fails when the processor on which that process is executing fails—once a process fails, it remains failed forever.[2] A distributed computation is *fault-tolerant* if it continues to execute correctly despite these failures. For simplification, we assume that every $p$ starts with a *dummy* event $e_{p,0}$ which is an initialization event. A distributed computation starts with each $p$ having executed $e_{p,0}$, and at any time later a process may *begin* (as we shall define) by executing a begin event, and after a begin event, it may *fail* (as we shall define) by executing a fail event [3].

The state of a process at any time is defined by its initial state after the dummy event, and by the sequence of events executed by it. We use notation $i, j$ to represent event identifiers on a process; $e_{p,i}$ to represent event number $i$ on $p$; and $m$, $m'$ to represent messages; $\mathcal{E}$ to denote the set of all non-internal events (including dummy events); and $\mathcal{E}_p$ to denote the set of all non-internal events (including dummy events) that occur on $p$. We assume that the computation is such that each non-dummy event in $\mathcal{E}$ is one of the following six types.

**0.** A *send* event, $s_{p,\mathcal{P},m}$, occurs at $p$ when $p$ sends a message $m$ to each process $q \in \mathcal{P}$. Once $p$ executes $s_{p,\mathcal{P},m}$, the system ensures that each $q \in \mathcal{P}$ receives $m$, unless $q$ fails. This implies that this message is received by only processes in a subset of $\mathcal{P}$.

**1.** A *receive* event, $r_{q,p,m}$, occurs at $q$ when $q$ receives a message $m$ from $p$.

**2.** A *begin* event, $b_p$, occurs at $p$ when $p$ begins the computation. This event is the first non-dummy event in $p$'s state, and occurs at most once. On its execution, $\mathcal{A}_p$, the set of processes that according to $p$ are *participating*, is initialized as $p$. Once $p$ executes $b_p$, the system ensures that each $q$ that has executed a $b_q$ (before $b_p$) and hasn't executed $f_q$, defined later, executes an $a_{q,\mathcal{P}}$, defined later, such that $p \in \mathcal{P}$.

**3.** An *add* event, $a_{q,\mathcal{P}}$, occurs at $q$ when $q$ takes note of the begin events $b_p$, for each $p \in \mathcal{P}$ for a non-empty $\mathcal{P}$. This event occurs at most once at $q$ for a given $p$. On its

[2] This ignores the scenarios where processes recover after a failure and rejoin the system. In later research we shall address how such rejoining can be incorporated in our model.

[3] Rather than assuming a model in which some other process *detects* failure of $p$ and executes a fail event on behalf of $p$, even though that is how fail event can be implemented, for simplicity we assume that $p$ executes fail event.

execution, $\mathcal{A}_q = \mathcal{A}_q \bigcup \mathcal{P}$ [4].

**4.** A *fail* event, $f_p$, occurs at $p$ when $p$ fails. This event occurs at most once at $p$, and is the last event in $p$'s state. We assume that such an event is executed by $p$ before it fails. Once $p$ executes $f_p$, the system ensures that each $q$ that has executed a $b_q$ (before $f_p$) and hasn't executed $f_q$, executes a $d_{q,\mathcal{P}}$ (as we shall define) such that $p \in \mathcal{P}$.

**5.** A *delete* event, $d_{q,\mathcal{P}}$, occurs at $q$ when $q$ takes note of the fail events $f_p$, for each $p \in \mathcal{P}$ for a non-empty $\mathcal{P}$. This event occurs at most once at $q$ for a given $p$. On its execution, $\mathcal{A}_q = \mathcal{A}_q - \mathcal{P}$.

Note that the first subscript of an event gives the process at which it occurs, the second subscript, if present, gives the other processes involved in the event,[5] and the third subscript, if present, gives the message sent or received. Whenever one or more subscripts from the right are unimportant, we drop them from our notation.

For our purposes internal events on a process can be ignored and so will not form a part of our formalism. Events in $\mathcal{E}$, other than the dummy events, can be divided into two groups; those which may cause other events, i.e. send, fail, and begin events, and those which are caused by some event in the first group, i.e. receive, delete, and add events. A process can spontaneously execute an event from the first group, and the underlying system causes a process to execute an event $e'$ from the second group if the event $e$ from the first group that causes $e'$ has been executed; this causal relationship between $e$ and $e'$ is denoted by $e \xrightarrow{c} e'$. This relationship suggests a definition of a relation "happened before" similar in spirit to (but more general than) the one defined in [10]. We redefine this relation for our model[6].

**Definition 1 Relation "Happened Before"** *denoted by* $\rightarrow$, *is the smallest relation on* $\mathcal{E}$ *such that* $e \rightarrow e''$ *if (0)* $e$ *and* $e''$ *happened on the same process and* $e$ *happened locally before* $e''$ *with respect to the process's clock, or (1)* $e \xrightarrow{c} e''$, *or (2) there exists an* $e'$ *in* $\mathcal{E}$ *such that* $e \rightarrow e'$ *and* $e' \rightarrow e''$. ∎

For our system, note the following:

- $\forall\, q \in \mathcal{P}$ such that $r_{q,p,m} \in \mathcal{E}_q$, $s_{p,\mathcal{P},m} \xrightarrow{c} r_{q,p,m}$,
- $\forall\, p \in \mathcal{P}$ and $\forall\, q \in \mathcal{D}$ such that $d_{q,\mathcal{P}} \in \mathcal{E}_q$, $f_p \xrightarrow{c} d_{q,\mathcal{P}}$,
- $\forall\, p \in \mathcal{P}$ and $\forall\, q \in \mathcal{D}$ such that $a_{q,\mathcal{P}} \in \mathcal{E}_q$, $b_p \xrightarrow{c} a_{q,\mathcal{P}}$.

In our model we assume that a mechanism exists that generates add and delete events at different processes caused by begin and fail events respectively. This mechanism is assumed to be live: it generates add and delete events eventually if no further failures occur. Examples of such mechanisms include those proposed in [12, 13]. This mechanism may satisfy certain properties about the set of processes where these events occur and the ordering of these events with respect to the other events. To make our model as general as possible, we assume this mechanism to satisfy only the following property, henceforth referred to as property **MP0**:

**MP0** $\forall\, b_p, e_p, a_{q,\mathcal{P}}$ such that $p \in \mathcal{P}$, if $\exists e_q : b_p \rightarrow e_p \wedge e_p \xrightarrow{c} e_q$ then $a_{q,\mathcal{P}} \rightarrow e_q$.

This property states that an add event at a process $q$ to include a process $p$ in $\mathcal{A}_q$ must happen before any event on $q$ that is caused by an event $e_p$ on $p$ such that $b_p$ happened before $e_p$. In addition to this property, mechanisms of [12, 13] also satisfy other properties. Effect of these other properties on our model is discussed in Section 6.2.2. Note that this mechanism is implemented by the underlying message communication system which in our model is not a part of the computation; this helps in achieving a clearer formalism and a clearer understanding.

# 3 A Global Snapshot in Fault-Tolerant Distributed Systems

Definitions of *global state* [5] and *global snapshot* [5] are modified as follows to incorporate events defined in Section 2. All discussion in this paper is for a fault-tolerant distributed system and so henceforth we will not explicitly mention it.

**Definition 2 A global state** *is defined to be a set of states*

- *one for each process in* $\mathcal{D}$ *such that for every* $e'$ *in a process's state and an* $e$ *such that* $e \xrightarrow{c} e'$, $e$ *is in the process state of the process where* $e$ *occurs.*
- *one for each channel* $c_{p,q}$ *corresponding to each* $q \in \mathcal{D}$ *whose state doesn't contain an* $f_q$, *as a set of messages sent along* $c_{p,q}$ *in* $p$*'s state minus the messages received by* $q$ *in* $q$*'s state.* ∎

**Definition 3 A global snapshot** *is a recorded global state.* ∎

# 4 Molecules and atoms

The essence of defining a molecule lies in noticing a cause and effect relationship for the new pairs of events, i.e. a fail event of a process and the corresponding delete events, and a begin event of a process and the corresponding add events, in addition to a pair for which this relationship has been well-studied, i.e. a send/multicast/broadcast event and the corresponding receive event(s) [5]. To facilitate stepwise understanding of our results, we define a number of terms, in addition to molecules and atoms; i.e. we define minimal dependence, general cause-and-effect or CE function, CE_Complete set, minimal CE_Complete set, wave, wavefront, wave_coverage or WC function, and ground state.

[4] Following C notation, we use '=' to represent assignment and '==' to represent equality

[5] This includes all processes to whom the message is sent in a send event, sender of the message in a receive event, and so on.

[6] Note that this definition is also applicable to the union of $\mathcal{E}$ and all internal events.

**Definition 4 Minimal dependence** *A unit of computation, which is a set of sequences of events, one each on each process in $\mathcal{P} \subseteq \mathcal{D}$, has minimal dependence on the rest of the computation if the computation of the unit is affected only by the states of processes in $\mathcal{P}$ at the start of the unit, and it affects the rest of the computation only by determining the states of processes in $\mathcal{P}$ at the end of the unit.* ∎

**Definition 5 General Cause-and-Effect** *or CE is a function that maps a dummy event to $\phi$, and a non-dummy event $e'$ to a set of events, possibly empty, that either are caused by $e'$, i.e. each $e''$ such that $e' \xrightarrow{c} e''$, or cause $e'$. So, $CE(e') = \{e : e' \xrightarrow{c} e \vee e \xrightarrow{c} e'\}$.* ∎

**Definition 5** *CE* **restated for our system** *For our system, using $\rightarrow$ for "happened before," we can express CE as follows:*
**0.** $CE(e_{p,0}) = \phi$
**1.** $CE(s_{p,\mathcal{P},m}) = \{r_{q,p,m} : \forall q \in \mathcal{P} : ((r_{q,p,m} \rightarrow f_q) \vee f_q \notin \mathcal{E}_q)\}$ [7]
**2.** $CE(r_{q,p,m}) = \{s_{p,\mathcal{P},m} : q \in \mathcal{P}\}$ [8]
**3.** $CE(f_p) = \{d_{q,\mathcal{P}} : \forall q : ((d_{q,\mathcal{P}} \rightarrow f_q \wedge p \in \mathcal{P}) \vee f_q \notin \mathcal{E}_q)\}$
**4.** $CE(d_{p,\mathcal{P}}) = \{f_q : q \in \mathcal{P}\}$
**5.** $CE(b_p) = \{a_{q,\mathcal{P}} : \forall q : ((a_{q,\mathcal{P}} \rightarrow f_q \wedge p \in \mathcal{P}) \vee f_q \notin \mathcal{E}_q)\}$
**6.** $CE(a_{p,\mathcal{P}}) = \{b_q : q \in \mathcal{P}\}$ ∎

**Definition 6 A CE-Complete set** *$\mathcal{S}$ is a non empty set of events such that $\forall e \in \mathcal{S}$, $CE(e) \subseteq \mathcal{S}$* ∎

**Definition 7 A minimal CE-Complete set** *is a CE-Complete set $\mathcal{E}$ such that no proper subset of $\mathcal{E}$ is CE-Complete.* ∎

Note that if $\mathcal{E}'$ and $\mathcal{E}''$ are CE-Complete sets, then $\mathcal{E}' \bigcup \mathcal{E}''$ and $\mathcal{E}' \bigcap \mathcal{E}''$ are also CE-Complete sets. Further, note that the minimal CE-Complete set containing an event $e$ is unique, and two minimal CE-Complete sets are either same or disjoint.

**Definition 8 A wave** *$\mathcal{W}$ is defined as follows:*
**0.** *$\mathcal{W}$ is a CE-Complete set.*
**1.** $\forall p \in \mathcal{D}$, $\exists e_{p,j} \in \mathcal{W}$.
**2.** $e_{p,j} \in \mathcal{W} \Rightarrow ((j = 0) \vee (e_{p,j-1} \in \mathcal{W}))$. ∎

**Definition 9 A wavefront** *$\mathcal{F}(\mathcal{W})$ is the set of last events, one on each process, of the wave $\mathcal{W}$.* ∎

[7] Note that $CE(s_{p,\mathcal{P},m})$ is empty if each $q \in \mathcal{P}$ fails before receiving $m$, i.e. $f_q \in \mathcal{E}_q$.
[8] $CE(r_{q,p,m})$ is always singleton.

**Definition 10 Wave_coverage or WC** *is a function that takes an event e and a given wavefront $\mathcal{F}(\mathcal{W})$ such that $e \notin \mathcal{W}$, denoted by $WC(e, \mathcal{F}(\mathcal{W}))$, and gives a set of events such that*
**1.** $e \in WC(e, \mathcal{F}(\mathcal{W}))$
**2.** *if $e' \in WC(e, \mathcal{F}(\mathcal{W}))$ then*
**(a)** *the minimal CE_Complete set containing $e'$ is a subset of $WC(e, \mathcal{F}(\mathcal{W}))$,*
**(b)** *$\forall e''$ such that $e'' \rightarrow e'$ and $e'' \notin \mathcal{W}$, $e'' \in WC(e, \mathcal{F}(\mathcal{W}))$.* ∎

Notice that in these definitions, a wavefront of the latest wave in the computation contains all fail events from processes that have failed, and dummy events from all processes that haven't yet begun. The execution of all events in a wave leaves the system in a *ground state*, as defined next.

**Definition 11 A ground state** *in a distributed system is defined to be a global state in which each process has executed its event in a wavefront; another equivalent definition is that it is a global state in which*
**0.** *for each p and q such that $f_q$ does not belong to the state of q in the global state, $c_{p,q}$ is empty,*
**1.** *for each $f_p$ event in the global state, all events in $CE(f_p)$ are included in the global state, and*
**2.** *for each $b_p$ event in the global state, all events in $CE(b_p)$ are included in the global state.* ∎

**Definition 12 A molecule** *is a set of events, $\mathcal{L}$, that satisfies the following properties:*
**A0** *For any p, if $e_{p,i}, e_{p,j} \in \mathcal{L}$, $j > i$, then $\forall k$ such that $i < k < j$, $e_{p,k} \in \mathcal{L}$.*
**A1** *$\mathcal{L}$ is a non-empty CE_Complete set.*
**A2** *$\mathcal{L}$ begins on a wavefront, $\mathcal{F}(\mathcal{W})$, i.e. $\exists \mathcal{W} : \forall e_{p,j} \in \mathcal{L} : ((e_{p,j-1} \in \mathcal{L}) \vee (e_{p,j-1} \in \mathcal{F}(\mathcal{W})))$.* ∎

A molecule has minimal dependence on other molecules, and its execution terminates on a wavefront or in a ground state.

**Definition 13 An Atom** *$l$ is a molecule such that no proper subset of $l$ is a molecule.* ∎

An atom has minimal dependence on other atoms or molecules, its execution (like a molecule) begins in a ground state and terminates in a ground state, and further it is *indivisible* (i.e. cannot be divided into smaller atoms). Figures 1 and 2 show some examples of waves, wavefronts, molecules, and atoms. In these figures, horizontal dotted arrows indicate the time line of processes, solid arrows indicate send and receive events, and dashed arrows indicate fail and delete events; begin and add events are not shown in this example, because they are similar to fail and delete events respectively.

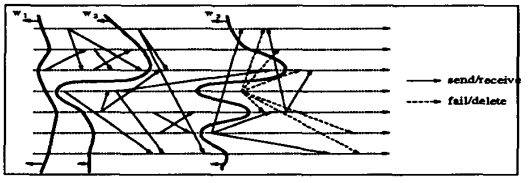


Figure 1: Some examples of waves and wavefronts in a distributed computation

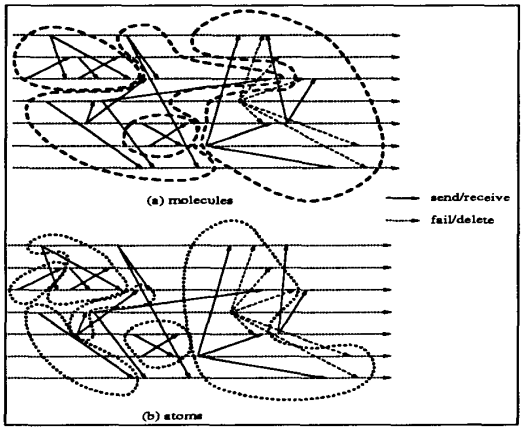


Figure 2: Some examples of (a) molecules, and (b) atoms in a distributed computation

**Definition 14 Occurs before relation** *A molecule $\mathcal{L}_1$ occurs before another molecule $\mathcal{L}_3$ iff*
- *for some p, $e_{p,j} \in \mathcal{L}_1$ and $e_{p,j+1} \in \mathcal{L}_3$, or*
- *$\exists$ a molecule $\mathcal{L}_2$ such that $\mathcal{L}_1$ occurs before $\mathcal{L}_2$, and $\mathcal{L}_2$ occurs before $\mathcal{L}_3$.* ∎

**Theorem 1** *Molecules can be ordered in a strict partial order using occurs before relation.* ∎

A new wave is formed every time an atom finishes its execution. So there is always one latest wave in the computation that includes all atoms that have finished execution, and the computation progresses from one ground state to next as this latest wave advances. Hence, the execution of atoms in a distributed computation captures the manner in which the computation progresses from one ground state to the next. While the actual execution of a distributed computation proceeds with the execution of individual events, it can also be viewed as an execution of atoms in the partial order defined by an occurs before relation.

By a simple record keeping, that is part of our system model, $p$ maintains $\mathcal{A}_p$, the set of processes that according to $p$ are *participating*, i.e. have executed a begin event but not a fail event. In a ground state, every $p$ will have the same $\mathcal{A}_p$. In Subsection 5.2, we will make use of this to determine the termination of an algorithm that forces a molecule and so a ground state in an ongoing execution. The following results, which we will use in the algorithm to detect atoms described in the next section, hold for wave_coverage and wavefront.

**Theorem 2** *$WC(e, \mathcal{F}(\mathcal{W}))$ is a molecule.* ∎

**Theorem 3** *A set $\mathcal{S}$ of events is a subset of some wavefront iff for each $e \in \mathcal{S}$, the condition* **C*** *described below, is satisfied for a given $\mathcal{F}(\mathcal{W})$.*

**C*** *: $\forall e' \in WC(e, \mathcal{F}(\mathcal{W}))$ and $e'' \in \mathcal{S}$, such that $e'$ and $e''$ occur on the same process, $e'' \not\rightarrow e'$.* ∎

# 5 Outline of Algorithms

Atoms, molecules, ground states, and waves in a computation occur as the computation progresses. From an application point of view, it would be important for a process to detect these, e.g. atoms, as they occur, and to force the completion of an ongoing molecule (and so a ground state) when desired. In this section we give outlines of the algorithms that do so. A detailed description of these algorithms is given in [3].

## 5.1 Detecting Atoms

The algorithm constructs an atom containing a given event *e_start*. The inputs to this algorithm are event *e_start* and some known wavefront $\mathcal{F}(\mathcal{W})$, such that *e_start* $\notin \mathcal{W}$. This algorithm starts by including the minimal CE_Complete set containing *e_start* in the set $\mathcal{U}$ (initially empty), and then repeats the following steps, and in the process adds new minimal CE_Complete sets to $\mathcal{U}$, until the construction of the atom is complete.

**S1** If all events in the boundary of $\mathcal{U}$ belong to some wavefront, then the construction of the atom is complete and the algorithm terminates.

**S2** Otherwise, if $e'$ is an event in the boundary of $\mathcal{U}$ that doesn't belong to any wavefront, then

**(i)** include in $\mathcal{U}$, the minimal CE_Complete set containing $e'$.

**(ii)** include in $\mathcal{U}$, the minimal CE_Complete set of each event $e''$ that satisfies (i) $e'' \notin \mathcal{U}$, and (ii) $\exists e_1, e_2 \in \mathcal{U}$ such that $e_1 \rightarrow e''$ and $e'' \rightarrow e_2$.

where boundary of a CE-Complete set $\mathcal{U}$ is the maximal set of events $e_{p,i}$ such that $e_{p,i} \notin \mathcal{U} \bigwedge e_{p,i+1} \in \mathcal{U}$. The algorithm uses Theorem 3 to determine if the events in set $\mathcal{U}$ belong to some wavefront.

This algorithm in the worst case may require examining every event in the computation, other than those in the wave $\mathcal{W}$, to detect an atom. This complexity, however, can be reduced if some additional conditions are known.

For example, if some other atoms in the computation are known, events belonging to those atoms or events that happened before any event in those atoms, need not be examined. If the communication pattern is restricted, as in synchronous communication or process group computation, all events need not be examined. We will discuss these in detail in Subsections 6.1 and 6.2.

## 5.2 Forcing a Ground state

While detecting atoms in general may involve examining entire set of events, we can create ground states more efficiently by forcing completion of the ongoing molecules. We give an outline of an algorithm to force a ground state assuming *F-channels*[9][1]. A process initiates this algorithm by sending f-markers to all processes in $\mathcal{A}_p$, where an *f-marker* has a property that it is always received after all messages sent before it, on that channels, have been received. This marker contains the set $\mathcal{A}_p$. On receiving an f-marker containing $\mathcal{A}_p$, a process $q$ sends f-markers if either this is the first f-marker received or if $\mathcal{A}_p - \mathcal{A}_q$ is not empty. The algorithm terminates at a process $p$ when for all processes $q \in \mathcal{A}_p$, $\mathcal{AS}[q] = \mathcal{A}_p$.

# 6 Understanding Computations

Dividing a fault-tolerant distributed computation into atoms and molecules helps in understanding the computation, because this allows one to concentrate on smaller parts of the computation in isolation. In general, more complicated the structure of an atom is, more difficult it is to understand the computation, except at the boundary of atoms and molecules. So, more complicated the structure permitted, more useful our formalism becomes. In this section, we examine two structures of atoms.

## 6.1 Communicating Sequential Processes

This model is proposed in [8]. In it only point-to-point communication is allowed and that too such that both sender and receiver must commit to a communication before it occurs. In such a computation, each minimal CE-Complete set consists of a send and the corresponding receive event, and every CE-Complete set in this model is a molecule. Figure 3(a) shows some of the atoms in this computation model.

In fact, the algorithm to force a ground state presented in Section 5 can be adapted easily to implement this communication model: adaptation requires that instead of using $\mathcal{A}_p$ and $\mathcal{A}_q$ in the algorithm, we use sets $\mathcal{S}_p$ and $\mathcal{S}_q$, in order to implement a synchronous communication between $p$ and $q$. $p$, in order to initiate a communication with $q$, executes this algorithm with the set $\mathcal{S}_p = \{p, q\}$. Similarly, $q$ executes this algorithm with the set $\mathcal{S}_q = \{p, q\}$. When the conditions for the recording rule are satisfied, $p$ and $q$ perform the actual communication.

[9]Since F-channels provide an ordering that is weaker than FIFO ordering, our algorithm will also work correctly if channels are FIFO.

## 6.2 Process Groups

A common way to structure a fault-tolerant distributed application is as a process group computation. In this model, a set of processes form a process group and communicate among themselves via a suitable multicast protocol; every message sent in the group is received by every member of the group. In addition, a membership protocol is used to maintain the membership of the group. It maintains a consistent system-wide view of group members by removing failed processes from the group membership, and incorporating new processes into the group membership. Process groups have been used in many distributed fault-tolerant systems [4, 12]. We first analyze the properties of atoms in this computation model without assuming any specific properties of the membership protocol, and then discuss the effect of specific properties of membership protocols.

### 6.2.1 Effect of Broadcast Communication

The communication among the group members is restricted to broadcast, and so every process in the group executes a receive event corresponding to every send event in the computation. The following result holds in a process group model.

**Theorem 4** *Atoms in a process group computation model, ordered according to occurs before relation form a total order.* ∎

Figure 3(b) shows some example atoms in this model. The structure of atoms is simplified, in that they contain events from all group members, by restricting all communication to broadcast. Also, since atoms are totally ordered, there is exactly one scenario that a given execution may take, and in this scenario the computation moves from one ground state to next by completing the execution of the next atom in the total order.

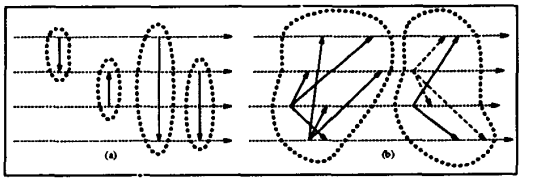


Figure 3: Examples of atoms in (a) communicating sequential processes model (b) process group model

The algorithm to detect atoms is also simplified in a process group model. Since an atom, and so also any minimal CE_Complete set, contains events from all processes in the group, the boundary of $\mathcal{U}$ contains an event for every process in the group. So, in order to determine if an

event $e$ satisfies **C***, only those events in WC($e$, $\mathcal{F}(W)$) that occur on the same process as $e$ and happened before $e$ need to be examined.

If the communication among group members is further restricted to *total order* broadcast, in which all broadcast messages are received in the same order at all group members [4, 6, 7], then the algorithm to detect atoms is further simplified by further simplification in determining if an event $e$ satisfies **C***. In this case, if condition **C*** is not satisfied for an event $e$, it will not be satisfied for any event that happened before $e$. Hence, the events that need to be examined are only those that belong to the minimal CE_Complete set containing $e$.

#### 6.2.2 Effect of Membership Protocols

We now turn our attention to the effect of specific membership protocols on atoms in a process group model. In particular, we focus our attention on the properties that these protocols satisfy with respect to the ordering of events in a distributed computation. We consider four such properties, one of which, i.e. **MP0**, was described in Section 2, and discuss protocols satisfying either all (as in [6, 13]) or only some (as in [12]) of the four. These properties are:

**MP0** $\forall\ b_p, e_p, a_{q,\mathcal{P}}$ such that $p \in \mathcal{P}$, if $\exists e_q : b_p \rightarrow e_p \wedge e_q \in CE(e_p)$ then $a_{q,\mathcal{P}} \rightarrow e_q$

This property implies that all events in the CE-Complete set of an event that happened after a begin event (i.e. the begin event happened before this event), happened after the corresponding add events at their respective processes.

**MP1** $\forall\ e_p, e_q, a_{p,\mathcal{P}}, a_{q,\mathcal{Q}}\ r \in \mathcal{P}, r \in \mathcal{Q}$, if $a_{p,\mathcal{P}} \rightarrow e_p \wedge e_q \in CE(e_p)$ then $a_{q,\mathcal{Q}} \rightarrow e_q$.

This property implies that all events in the CE-Complete set of an event that happened after an add event (i.e. the add event happened before this event), happened after the corresponding add events at their respective processes.

**MP2** $\forall\ f_p, e_p, d_{q,\mathcal{P}}$ such that $p \in \mathcal{P}$, if $\exists e_q : e_p \rightarrow f_p \wedge e_q \in CE(e_p)$ then $e_q \rightarrow d_{q,\mathcal{P}}$.

This property implies that all events in the CE-Complete set of an event that happened before a fail event, happened before the corresponding delete events at their respective processes.

**MP3** $\forall\ e_p, e_q, d_{p,\mathcal{P}}, d_{q,\mathcal{Q}}$ such that $r \in \mathcal{P}$, $r \in \mathcal{Q}$, if $d_{p,\mathcal{P}} \rightarrow e_p \wedge e_q \in CE(e_p)$ then $d_{q,\mathcal{Q}} \rightarrow e_q$.

This property implies that all events in the CE-Complete set of an event that happened after a delete event (i.e. the delete event happened before this event), happened after the corresponding delete events at their respective processes.

**Theorem 5** *When Properties* **MP0**, **MP1**, **MP2**, *and* **MP3** *are satisfied,*
*(A)* $f_p \bigcup CE(f_p) \bigcup \{$*fail events on processes on which no event occurs in* $CE(f_p)\}$ *form a wavefront.*
*(B)* $b_p \bigcup CE(b_p) \bigcup \{$*fail events on processes on which no event occurs in* $CE(b_p)\}$ *form a wavefront.* ∎

The membership protocols proposed in [6, 13] satisfy all these properties. Thus, these protocols create a wavefront in the execution whenever a process executes a fail event or a begin event. When these protocols terminate, the system is in a ground state. Hence, the events corresponding to a change in the group membership occur at a ground state. On the other hand, the membership protocol proposed in [12] does not satisfy property **MP3**. So when this membership protocol terminates, and when changes to the membership are done, the system need not be in a ground state. An advantage of this is that there is no need to force a ground state in order to implement this protocol, and so the cost of the protocol is less. Indeed this advantage comes at the cost of a more complicated structure of the atoms that result from this protocol.

## 7 Applications of this work

There are several applications of this work. These applications have been well studied for system models that assume only point-to-point message communication and/or no process failures. These applications can be generalized to our model and in general this generalization may include any pair of cause and effect events.

Even though we defined waves and wavefronts to lead us into definitions of molecules and atoms, waves and wavefronts are useful abstractions in themselves. A wave is useful as it is left-closed in the sense that if an event is in a wave so is every other event that happened before it. A wavefront is useful as it has only the last event in a wave on each process, as is often needed in reasoning. Ground state is useful in stable property [5] detection particularly for stable properties such as termination detection[15], and in checkpointing process states without any need for process rollbacks on recovery.

There are several applications of the algorithm for forcing a ground state, in addition to implementing synchronous communication in communicating sequential processes communication model. The algorithm can be used in the implementation of membership protocols. In order to implement add and delete events, the membership protocol simply forces a ground state whenever begin (or fail) events occur and then before executing any other event, add (or delete) events are executed at each process that has not failed. This algorithm can also be used to do consistent checkpointing.

As we mentioned before, definition of a molecule is essentially a generalization and formalization (of this generalization) of an atomic action [11]. So applications of our work include those of atomic actions (for the generalized model), e.g. in checkpointing and recovery. It is worth noting that models for process checkpointing and recovery have been developed in [14, 9]. Our model differs from these in many aspects. First, these models consider only two types of events: send and receive, while

our model incorporates other types of events such as fail, delete, begin, add, and multicast, and in fact it provides a mechanism to include any other types of events based on cause and effect relationship. Second, these models define *state intervals* based on receive events occurring only on a single process, while the unit of computation in our model is based on cause and effect relationship between events occurring on multiple processes; this we believe gives a more general model. Finally, these models are specific to process checkpointing and recovery while our model is a more general model that can be used in several applications such as program debugging, stable property detection and so on, in addition to process checkpointing and recovery.

An atom is an indivisible molecule and so reasoning gets further stronger. Detection of atoms can be used in debugging programs. In order to determine the cause of a bug in a program, it is important to first understand how different events in the computation are affected by other events. However, when processes in a computation interact asynchronously, and may fail at any time, it is not a single event but a logical group of events that collectively affect other logical groups of events. A debugging technique based on such logical groups of events is certainly more meaningful. Clearly, such a logical group should be as small as possible for simplicity, and atoms are the smallest such groups. Debugging based on atoms in a computation would involve debugging in two stages. The first stage involves finding the latest atoms in the occur before relation after whose execution the observed problem first manifested; and then in the second stage the event(s) within the atom are observed to determine the event after whose execution the observed problem first manifested.

Atoms and molecules also have applications in designing programming languages and the needed support. While atoms and molecules are created as the execution of a computation proceeds, it would be more useful if the programmer could control the creation of atoms and molecules in such computations. This would simplify writing programs for such computations by allowing a programmer to concentrate on smaller logical parts of the computation in isolation without worrying about any interference from the rest of the computation. For example, an atomic action, in such a system, can be programmed as follows: the atomic action is written as a molecule and checkpoints are taken at the beginning and end of the molecule. If a failure occurs during the execution of a molecule, all processes are simply rolled back to their last checkpoints.

# 8 Conclusion

We have developed a framework that helps in understanding a fault-tolerant distributed system and so helps in designing such systems. There are two areas in which this work needs to be extended. First, we would like to relax our assumption of a reliable communication network, and thus consider message losses: to do so, definitions will need appropriate modifications. Second, we would like to consider recovery of processes: this will require definition and inclusion of some new types of events. Work is in progress in these areas.

# References


[1] M. Ahuja. Flush primitives for asynchronous distributed systems. *Information Processing Letters*, 34(1):5–12, 1990.

[2] M. Ahuja, A. D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *Proceedings of Tenth International Conference on Distributed Computing Systems*, pages 12–19, May 1990.

[3] M. Ahuja and S. Mishra. A basic unit of computation in fault-tolerant distributed systems. Technical Report CS93-277, Dept. of Computer Science and Engineering, University of California, San Diego, CA, 1993.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[5] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

[6] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[7] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[8] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[9] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, pages 462–491, 1990.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[11] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[12] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, Dec 1993.

[13] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Eleventh ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1991.

[14] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug 1985.

[15] A. J. M. van Gastern, E. W. Dijkstra, and W. H. J. Feijen. Derivation of termination detection algorithm for distributed computation. *Information Processing Letters*, 16:217–219, 1983.