Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander, Phil Miller, Laxmikant V. Kale
{jliff12, mille121, kale}@illinois.edu

University of Illinois Urbana-Champaign

February 26, 2013
Adoption Protocols for

Fanout-Optimal

Fault-Tolerant

Termination Detection
Adoption Protocols for

Fanout-Optimal

Fault-Tolerant

Termination Detection
Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

What is it?

Algorithm overview
Adoption Protocols for

Fanout-Optimal

Fault-Tolerant

Termination Detection

What is it?
Why is it relevant?
Adoption Protocols for

Fanout-Optimal

Fault-Tolerant

Termination Detection

What is it?
Why is it relevant?
Algorithm overview
Adoption Protocols for Fanout-Optimal Fault-Tolerant

Problem description

Algorithm overview

Fault-Tolerant

Termination Detection

What is it?

Why is it relevant?

Algorithm overview
Adoption Protocols for Fanout-Optimal Fault-Tolerant

Problem description

Previous work

What is it?

Why is it relevant?

Algorithm overview

Fault-Tolerant

Termination Detection
Adoption Protocols for

Fanout-Optimal $\rightarrow$ Theoretical bounds

Fault-Tolerant $\rightarrow$ Problem description
Previous work

Termination Detection $\rightarrow$ What is it?
Why is it relevant?
Algorithm overview
Adoption Protocols for

Fanout-Optimal

Fault-Tolerant

Termination Detection

Trio of protocols:

INDEP, RELAZY, RELAGER

Probabilistic model

High survivability

Theoretical bounds

Problem description

Previous work

What is it?

Why is it relevant?

Algorithm overview
Adoption Protocols for

Trio of protocols: \{ \text{INDEP, RELAZYE, RELEAGER} \}

Fanout-Optimal

Theoretical bounds

Fault-Tolerant

Problem description
Previous work

Termination Detection

What is it?
Why is it relevant?
Algorithm overview
Adoption Protocols for Trio of protocols: \{INDEP, RELAZY, RELEAGER\}

Probabilistic model

Fanout-Optimal Theoretical bounds

Fault-Tolerant Problem description

Previous work

Termination Detection What is it?

Why is it relevant?

Algorithm overview
Adoption Protocols for

Trio of protocols: \{\text{INDEP, RELAZY, RELEAGER}\}

Probabilistic model
High survivability

Fanout-Optimal

Theoretical bounds

Fault-Tolerant

Problem description
Previous work

Termination Detection

What is it?
Why is it relevant?
Algorithm overview
Adoption Protocols for

Trio of protocols: \{INDEP, RELAZY, RELEAGER\}

Probabilistic model
High survivability
Empirical Results

Fanout-Optimal

Theoretical bounds

Fault-Tolerant

Problem description
Previous work

Termination Detection

What is it?
Why is it relevant?
Algorithm overview
Termination Detection

What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight
Termination Detection

→ What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
Termination Detection

What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
  - A process is either *active* or *passive*
Termination Detection

What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
  - A process is either active or passive
  - A process starts passive (except for the root)
Termination Detection

What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

Fundamental assumptions
- A process is any schedulable entity: thread, object, etc.
- A process is either active or passive
- A process starts passive (except for the root)
- A process only becomes active when it receives a message
Termination Detection

→ What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
  - A process is either active or passive
  - A process starts passive (except for the root)
  - A process only becomes active when it receives a message
  - A process can only send messages in the active state
Termination Detection

→ What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
  - A process is either active or passive
  - A process starts passive (except for the root)
  - A process only becomes active when it receives a message
  - A process can only send messages in the active state
  - A process transitions back to passive after processing a message
Termination Detection

What is it?

- The “global” system state when
  - all processes are idle and
  - no messages are in flight

- Fundamental assumptions
  - A process is any schedulable entity: thread, object, etc.
  - A process is either active or passive
  - A process starts passive (except for the root)
  - A process only becomes active when it receives a message
  - A process can only send messages in the active state
  - A process transitions back to passive after processing a message
  - idle ≡ passive
Termination Detection

Why is it relevant?

- As parallel computations become more dynamic, detecting termination is non-trivial
Termination Detection

→ Why is it relevant?

- As parallel computations become more dynamic, detecting termination is non-trivial
  - Adaptive mesh refinement*

Termination Detection
→ Why is it relevant?

- As parallel computations become more dynamic, detecting termination is non-trivial
  - Adaptive mesh refinement*
  - Dynamic data exchanges (e.g. SPMV)

Termination Detection

→ Why is it relevant?

- As parallel computations become more dynamic, detecting termination is non-trivial
  - Adaptive mesh refinement*
  - Dynamic data exchanges (e.g. SPMV)
  - Distributed-memory work stealing – task scheduling†


Termination Detection

Why is it relevant?

- As parallel computations become more dynamic, detecting termination is non-trivial
  - Adaptive mesh refinement*
  - Dynamic data exchanges (e.g. SPMV)
  - Distributed-memory work stealing – task scheduling†
  - Runtimes with message-driven execution‡

---


Many different variants (with different tradeoffs)

- Parental responsibility
- Wave-based
- Credit-recovery

---

Termination Detection

→ Algorithm Overview: Parental Responsibility

\[ \{ c = 0 ; d = 0 \} \]

\[ p_0 \]

\[ p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_{n-1} \]
Termination Detection

→ Algorithm Overview: Parental Responsibility

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander

\[ \{ c = 0 \; ; \; d = 3 \} \]
Termination Detection

Algorithm Overview: Parental Responsibility

- $p_0$: $c = 0; d = 3$
- $p_2$: $c = 1, d = 0$
- $p_0$
- $p_1$
- $p_3$
- $p_4$
- $p_5$
- $p_{n-1}$

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander
Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Termination Detection

→ Algorithm Overview: Parental Responsibility

$$\begin{align*}
\{ c = 0 ; d = 3 \} \\
\{ c = 1 , d = 0 \} \\
\{ p_0 \} \\
\{ c = 1 , d = 0 \} \\
\{ p_0 \}
\end{align*}$$
Termination Detection

Algorithm Overview: Parental Responsibility

\[
\begin{align*}
\{ c = 0 ; d = 3 \} \\
\{ c = 1, d = 1 \} & \quad \{ p_0 \} \\
\{ c = 1, d = 0 \} & \quad \{ p_0 \}
\end{align*}
\]
Termination Detection

Algorithm Overview: Parental Responsibility

Algorithm:

1. Initial state: \( \{ c = 0; d = 3 \} \)
2. \( p_0 \) checks its parent.
3. \( p_0 \) sends a message to \( p_2 \) with \( c = 1, d = 1 \).
4. \( p_2 \) receives the message and checks its parent.
5. \( p_2 \) sends a message to \( p_4 \) with \( c = 1, d = 0 \).
6. \( p_4 \) receives the message and checks its parent.
7. \( p_4 \) sends a message to \( p_5 \) with \( c = 1, d = 0 \).
8. \( p_5 \) receives the message and checks its parent.
9. \( p_5 \) sends a message to \( p_0 \) with \( c = 1, d = 0 \).
10. \( p_0 \) receives the message and checks its parent.
11. \( p_0 \) sends a message to \( p_2 \) with \( c = 1, d = 0 \).

Key:
- \( p_0 \) is the root node.
- \( p_1, p_2, p_3, p_4, p_5 \) are children of \( p_0 \).
- \( p_n \) is the \( n \)-th child of \( p_0 \).
Termination Detection

→ Algorithm Overview: Parental Responsibility

\[ \{ c = 0 ; d = 3 \} \]
\[ \{ c = 1, d = 1 \} \]
\[ \{ p_0 \} \]
\[ \{ c = 1, d = 0 \} \]
\[ \{ p_0 \} \]
\[ \{ c = 2, d = 0 \} \]
\[ \{ p_2 ; p_0 \} \]
Termination Detection

→ Algorithm Overview: Parental Responsibility

Diagram:

- $p_0$ with $c = 0; d = 3$
- $p_2$ with $c = 1; d = 1$
- $p_4$ with $c = 1; d = 0$
- $p_5$ with $c = 1; d = 0$
- $p_1$, $p_3$, $p_{n-1}$
Termination Detection

Algorithm Overview: Parental Responsibility

\[ p_0 \]
\{ \( c = 0 \); \( d = 2 \) \}

\[ p_2 \]
\{ \( c = 1 \), \( d = 1 \) \}
\{ \( p_0 \) \}

\[ p_4 \]
\{ \( c = 1 \), \( d = 0 \) \}
\{ \( p_2 \) \}

\[ p_5 \]
\{ \( c = 1 \), \( d = 0 \) \}
\{ \( p_0 \) \}

\[ p_1 \]

\[ p_3 \]

\[ p_{n-1} \]
Termination Detection

Algorithm Overview: Parental Responsibility

\[
\begin{align*}
\{ c = 0 ; d = 2 \} \\
\{ c = 1 , d = 1 \} \\
\{ p_0 \} \\
\{ c = 0 , d = 0 \} \\
\{ } \\
\{ c = 1 , d = 0 \} \\
\{ p_2 \} \\
ack
\end{align*}
\]
Termination Detection

→ Algorithm Overview: Parental Responsibility

\[
\begin{align*}
{c = 0; d = 1} & \quad \{ p_0 \} \\
{c = 1, d = 1} & \quad \{ p_2 \} \\
{c = 1, d = 0} & \quad \{ p_4 \} \\
{c = 1, d = 0} & \quad \{ p_5 \} \\
{c = 1, d = 0} & \quad \{ p_{n-1} \}
\end{align*}
\]
Termination Detection

Algorithm Overview: Parental Responsibility
Termination Detection

Algorithm Overview: Parental Responsibility

\[
\begin{align*}
p_0 & \{ c = 0 ; d = 1 \} \\
p_2 & \{ c = 1, d = 0 \} \\
p_1 & \\
p_3 & \\
p_4 & \\
p_5 & \\
p_{n-1} & 
\end{align*}
\]
Termination Detection

→ Algorithm Overview: Parental Responsibility

\[ \{ c = 0; d = 1 \} \]

\[ \{ c = 0, d = 0 \} \]

\[ \{ \} \]

\[ \text{ack} \]

\[ p_0 \]

\[ p_2 \]

\[ p_1 \]

\[ p_3 \]

\[ p_4 \]

\[ p_5 \]

\[ p_{n-1} \]
Termination Detection

→ Algorithm Overview: Parental Responsibility

Termination Detected!

\[ p_0 \{ c = 0 ; d = 0 \} \]

Termination Detected!

\[ p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_{n-1} \]
git://charm.cs.illinois.edu/terminator.git
git://charm.cs.illinois.edu/terminator.git

General C++ and Java library
git://charm.cs.illinois.edu/terminator.git

Implemented in three parallel runtime systems
Being made fault-tolerant based on this work
Fault-Tolerant Termination Detection

Problem Description

- Approaches to fault-tolerance
  - General runtime-system support: checkpointing, message-logging, etc.
Fault-Tolerant Termination Detection

→ Problem Description

- Approaches to fault-tolerance
  - General runtime-system support: checkpointing, message-logging, etc.
  - Algorithm-specific (checksum-based approaches for math libraries)

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander
Approaches to fault-tolerance

- General runtime-system support: checkpointing, message-logging, etc.
- Algorithm-specific (checksum-based approaches for math libraries)
- Component-specific: runtime system is composed of a set of self-healing components that all handle faults in their own optimized way — so-called *scale invariance*.

---

Fault-Tolerant Termination Detection

→ Component-specific

- Assume ecosystem is fault tolerant
  - Application can recover from faults
  - Other runtime system components are fault-tolerant
  - Termination can be handled as a modular component
Fault-Tolerant Termination Detection

Previous work

- Distributed computing
  - An \((n-1)\)-resilient algorithm for distributed termination detection.

Fault-Tolerant Termination Detection

Previous work

- Distributed computing
  - An \((n-1)\)-resilient algorithm for distributed termination detection.
  - Recovers from up to \(n-1\) faults in the system!

---

Fault-Tolerant Termination Detection

→ Previous work

- Distributed computing
  - An \((n-1)\)-resilient algorithm for distributed termination detection.
  - Recovers from up to \(n-1\) faults in the system!
  - But not practical for HPC

---

Fault-Tolerant Termination Detection

→ Previous work

- Distributed computing
  - An \((n-1)\)-resilient algorithm for distributed termination detection.
  - Recovers from up to \(n-1\) faults in the system!
  - But not practical for HPC
  - Serializes recovery through the root (one process) of the computation

---

Fault-Tolerant Termination Detection

Previous work

- Distributed computing
  - An \((n-1)\)-resilient algorithm for distributed termination detection.
  - Recovers from up to \(n-1\) faults in the system!
  - But not practical for HPC
  - Serializes recovery through the root (one process) of the computation
  - Requires all processes (even unengaged) to communicate (obviating many beneficial properties of parental responsibility algorithms)

Distributed computing

- An \((n-1)\)-resilient algorithm for distributed termination detection.
- Recovers from up to \(n-1\) faults in the system!
- But not practical for HPC
- Serializes recovery through the root (one process) of the computation
- Requires all processes (even unengaged) to communicate (obviating many beneficial properties of parental responsibility algorithms)
- Has low overhead on forward execution

Fault-Tolerant Termination Detection

→ Goals for HPC

- Low overhead during forward execution
- Expect to encounter faults that only impact a small subset of the processes
- Build a scalable algorithm for recovery
Parental responsibility algorithms

- **Message-optimal**: in the worst case, they send the lower-bound on signal count**
- Where the lower bound is $O(m)$, and $m$ is the total number of application messages

Termination Detection

Optimality

- **Fanout-optimality**
  - The *fanout* or $f$ for a given process is the number of communication partners it has.
Termination Detection

→ Optimality

- **Fanout-optimality**
  - The *fanout* or $f$ for a given process is the number of communication partners it has
  - Because it follows the communication graph of the application, it will be as scalable as the application
Termination Detection

→ Optimality

- **Fanout-optimality**
  - The *fanout* or $f$ for a given process is the number of communication partners it has
  - Because it follows the communication graph of the application, it will be as scalable as the application
  - For parental-responsibility algorithms, this is a dynamic property
Termination Detection

→ Optimality

■ Fanout-optimality
  ▶ The *fanout* or *f* for a given process is the number of communication partners it has
  ▶ Because it follows the communication graph of the application, it will be as scalable as the application
  ▶ For parental-responsibility algorithms, this is a dynamic property
  ▶ Optimality: in the worst case, the algorithm sends $O(f)$ signals
**Termination Detection**

→ **Optimality**

- **Fanout-optimality**
  - The *fanout* or $f$ for a given process is the number of communication partners it has
  - Because it follows the communication graph of the application, it will be as scalable as the application
  - For parental-responsibility algorithms, this is a dynamic property
  - Optimality: in the worst case, the algorithm sends $O(f)$ signals
  - The fault-tolerance algorithms we present are fanout-optimal
Termination Detection

→ Fault-tolerance assumptions

- General
  - *Fail-stop* model: failed processes do not recover
Termination Detection

→ Fault-tolerance assumptions

- General
  - *Fail-stop* model: failed processes do not recover
  - External system (or runtime system) provides failure notification
Termination Detection

Fault-tolerance assumptions

- **General**
  - *Fail-stop* model: failed processes do not recover
  - External system (or runtime system) provides failure notification
  - No byzantine failures: failed processes do not behave maliciously
Termination Detection

→ Fault-tolerance assumptions

- General
  - *Fail-stop* model: failed processes do not recover
  - External system (or runtime system) provides failure notification
  - No byzantine failures: failed processes do not behave maliciously

- Specific
Termination Detection

→ Fault-tolerance assumptions

- General
  - *Fail-stop* model: failed processes do not recover
  - External system (or runtime system) provides failure notification
  - No byzantine failures: failed processes do not behave maliciously

- Specific
  - *Network send fence*: messages are “on-the-wire”
Termination Detection

→ Fault-tolerance assumptions

- General
  - *Fail-stop* model: failed processes do not recover
  - External system (or runtime system) provides failure notification
  - No byzantine failures: failed processes do not behave maliciously

- Specific
  - Network send fence: messages are “on-the-wire”
  - Fail-flush: a system notification indicating that in-flight messages from a failed process have all been received
Termination Detection

The INDEP fault-tolerance protocol
Termination Detection

The INDEP fault-tolerance protocol
Termination Detection

The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection

The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection
→ The INDEP fault-tolerance protocol
Termination Detection

→ The **INDEP** fault-tolerance protocol

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander

19 / 25
Termination Detection

The INDEP fault-tolerance protocol
Termination Detection

The INDEP fault-tolerance protocol

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection
Jonathan Lifflander
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol
Termination Detection

→ The INDEP fault-tolerance protocol

- Tolerates all single-process failures

What multi-process failures does it not tolerate?

▶ INDEP will tolerate all failures except for parent-child pairs...

▶ But, INDEP cannot detect this case, so it has to fail conservatively if the failure set has communicating pairs in it
Termination Detection

- The INDEP fault-tolerance protocol

- Tolerates all single-process failures
- What multi-process failures does it not tolerate?
Termination Detection

→ The INDEP fault-tolerance protocol

- Tolerates all single-process failures
- What multi-process failures does it not tolerate?
  - INDEP will tolerate all failures except for parent-child pairs...
Termination Detection

→ The INDEP fault-tolerance protocol

- Tolerates all single-process failures
- What multi-process failures does it not tolerate?
  ▶ INDEP will tolerate all failures except for parent-child pairs...
  ▶ But, INDEP cannot detect this case, so it has to fail conservatively if the failure set has communicating pairs in it
Termination Detection

→ The \texttt{INDEP} fault-tolerance protocol
Termination Detection

→ The RelLazy and RelEager protocols
Termination Detection

→ The **ReLLazy** and **ReLEager** protocols
Termination Detection

→ The RelLazy and RelEager protocols

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander
Termination Detection

\[ \text{The \textsc{RelLazy} and \textsc{RelEager} protocols} \]
Termination Detection

→ The ReLLazy and ReLEager protocols

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander
Termination Detection

→ The RelLazy and RelEager protocols

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander

22 / 25
Termination Detection

→ The RelLazy and RelEager protocols
Termination Detection

→ The RelLazy and RelEager protocols
Termination Detection

→ The RelLazy and RelEager protocols

Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection

Jonathan Lifflander

22 / 25
## Termination Detection

→ **Probability Model and Survivability**

<table>
<thead>
<tr>
<th>Nodes Failed</th>
<th>Fault (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>92.30</td>
</tr>
<tr>
<td>2</td>
<td>3.672</td>
</tr>
<tr>
<td>3</td>
<td>0.942</td>
</tr>
<tr>
<td>4</td>
<td>0.753</td>
</tr>
<tr>
<td>5</td>
<td>0.565</td>
</tr>
<tr>
<td>6</td>
<td>0.094</td>
</tr>
<tr>
<td>7</td>
<td>0.094</td>
</tr>
<tr>
<td>8</td>
<td>0.377</td>
</tr>
<tr>
<td>9</td>
<td>0.094</td>
</tr>
<tr>
<td>10</td>
<td>0.188</td>
</tr>
<tr>
<td>11</td>
<td>0.188</td>
</tr>
<tr>
<td>15</td>
<td>0.282</td>
</tr>
<tr>
<td>18</td>
<td>0.094</td>
</tr>
<tr>
<td>26</td>
<td>0.094</td>
</tr>
<tr>
<td>86</td>
<td>0.094</td>
</tr>
<tr>
<td>126</td>
<td>0.094</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Survivability* (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEP</td>
<td></td>
</tr>
<tr>
<td>( f = 2 )</td>
<td>99.32</td>
</tr>
<tr>
<td>( f = 8 )</td>
<td>98.63</td>
</tr>
<tr>
<td>( f = 32 )</td>
<td>97.47</td>
</tr>
<tr>
<td>( f = 512 )</td>
<td>93.21</td>
</tr>
<tr>
<td>REL*</td>
<td>99.50</td>
</tr>
</tbody>
</table>

*Assuming a 1024-node job
Termination Detection

→ Empirical Results

- Ratio of execution time without FT compared to using the REL* protocols
- Sample size of 24, using a Student’s t-test, error bars represent standard error at 99% confidence
- Using distributed-memory work stealing — NQueens (NQ), HF (Hartree-Fock), TCE (Tensor Contraction Expressions)
Termination Detected!

git://charm.cs.illinois.edu/terminator.git