Deadlock Avoidance for Distributed Real-Time and Embedded Systems

Henny Sipma
Stanford University

Joint work with
Cesar Sanchez, Zohar Manna,
Venkita Subramonian, Chris Gill

Washington University at St Louis
Nov 17, 2006
Goal: Formalization of Middleware Services

Middleware

- Real-time Event Channel
- Event Filtering
- Scheduler

Deadlock avoidance

OS1  OS2  OS3  OS4  OS5  OS6
Deadlock is a classical problem in Computer Science

Deadlock is the situation in which resources have been allocated to various tasks in such a way that none of the tasks can continue.

Habermann, 1969

Classical example: Dining philosophers
Deadlock avoidance in DRE systems

Deadlock

Conditions:
1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait
Deadlock Prevention

Approach:
- Statically break the circular wait

Disadvantages:
- Sacrifices concurrency and reduces resource utilization
- Burden on the programmer

Deadlock prevention in DRE systems
Solutions: Deadlock Detection

Approach

Make one of the components release its resources and go back to an earlier state

Problem:

Common in databases, but not practical in embedded systems
Solutions: Deadlock Avoidance

Approach:

Dynamically assign resources on request based on availability and other processes’ needs

Dijkstra’s Banker’s algorithm
Deadlock avoidance in DRE systems

### Deadlock avoidance problem space

- **Centralized**
  - Unsolvable
  - Max utilization
    - [Dijkstra, 65]
  - Call sequences
    - [deAlfaro, 05]
    - FMS

- **Distributed**
  - Unsolvable
  - Impractical
    - [Singhal, 95]
  - Call sequences
    - [Sanchez++, 05, 06]
## Solutions

<table>
<thead>
<tr>
<th>method</th>
<th>centralized</th>
<th>distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>prevention</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>detection</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>avoidance</td>
<td>Banker’s algorithm: maximum # resources</td>
<td>impractical in general [Singhal95]</td>
</tr>
<tr>
<td></td>
<td>Flexible manufacturing systems: # resources + order of request</td>
<td>practical with more a priori knowledge of process structure: Call Graphs [Sanchez++ 05,06,06,06]</td>
</tr>
</tbody>
</table>
Design a deadlock avoidance protocol that

- is local to each component (does not require any communication between components)
- makes use of static information (call graphs)
- allows maximum concurrency
- can be implemented efficiently
OUTLINE

- Introduction
- Deadlock avoidance protocols
- Liveness
- Distributed priority inheritance
- Related Work
- Conclusions
distributed components
A, B, C, D, E

Sequence of calls

Deadlock avoidance in DRE systems
Assumptions:

- Asynchronous distributed system
- Limited resources
- Remote method invocations
- WaitOnConnection policy
- Arbitrary number of processes spawned
- All processes terminate

Resources are threads or execution contexts
EXAMPLE OF DEADLOCK

Two components with two threads each

Two call graphs

Deadlock avoidance in DRE systems

Washington University, November 17, 2006
Distributed system model

**S: \( \langle R, G \rangle \)**

**R: \{A, B, \ldots\}, a set of components.** Each component \( A \) maintains local variables including

- \( T_A: \) (constant) total number of threads
- \( t_A: \) number of threads currently available

**G: \{G_1, G_2, \ldots\}, a set of acyclic call graphs**
OUTLINE

- **Introduction**
- **Deadlock avoidance protocols**
- **Liveness**
- **Distributed priority inheritance**
- Related Work
- Conclusions
Deadlock avoidance protocol

Protocol schema: (executed for every incoming method call)

- **Enabled\(_n\)(V\(_A\))**: condition evaluated over local variables of A
  - if true: a thread can be granted
- **In\(_n\)(V\(_A\),V\(_A\)'\):** update to local variables if thread is granted
- **Out\(_n\)(V\(_A\),V\(_A\)'\):** update to local variables when thread is released

```
when Enabled\(_n\)(V\(_A\)) do
  In\(_n\)(V\(_A\),V\(_A\)'\)

n\(_A\)::
  f()

Out\(_n\)(V\(_A\),V\(_A\)'\)
```

entry section

method invocation

exit section
Global call graph: \( G = G_1 \cup G_2 \cup \ldots = \langle \mathcal{V}, \mathcal{E} \rangle \)

Annotation \( \alpha : \mathcal{V} \rightarrow \mathbb{N} \), mapping nodes to natural numbers

annotation represents the minimum number of threads that should be reserved for other processes (executing the same or other call graph)
**PROTOCOL basic-P**

\[
\text{when } \text{Enabled}_n(V_A) \text{ do }
\]
\[
\text{In}_n(V_A, V_{A'})
\]

\[
\text{entry section}
\]

\[
\text{n}_A::
\]
\[
f()
\]

\[
\text{method invocation}
\]

\[
\text{Out}_n(V_A, V_{A'})
\]

\[
\text{exit section}
\]

\[
\text{Enabled}_n(V_A) : t_A > \alpha(n)
\]

\[
\text{In}_n(V_A, V_{A'}) : t_A++
\]

\[
\text{Out}_n(V_A, V_{A'}) : t_A--
\]

\[t_A: \text{number of currently available threads}\]
Question:

What conditions must $\alpha$ satisfy to ensure deadlock freedom?
Answer: \( \alpha \) must have no cyclic dependencies

Example:

Node \( v \) directly depends on node \( w \) if

- \( v \) makes a method call to \( w \), or
- \( v, w \) run in the same component and \( \alpha(v) \geq \alpha(w) \)

Node \( v \) depends on node \( w \) if

- there is a path from \( v \) to \( w \) that includes at least one

\( \alpha \) is acyclic if

- no node depends on itself
Theorem
Let $\alpha$ be an annotation of the global call graph. If Basic-P is used in all components to decide allocation of threads, and $\alpha$ is acyclic, then the system is deadlock free.
Annotation Theorem provides a **sufficient** condition for deadlock freedom.

Question: Is it also a **necessary** condition?

**Theorem**

Let $\alpha$ be an annotation of the global call graph. If Basic-P is used in all components to decide allocation of threads, and $\alpha$ is not acyclic, then, given sufficient resources, a **deadlock is always reachable**.
No deadlocks are reachable with $T_C = 1$

But, deadlocks are reachable with $T_C = 2$

Increasing the total number of threads may turn a deadlock-free system into a system that is not deadlock-free

Checking whether a system with a cyclic annotation is deadlock-free is NP-complete
How to compute an acyclic annotation?

- Simplest solution: height in the graph

Minimum number of threads required:

- $T_A = 4$
- $T_B = 2$
- $T_C = 3$
- $T_D = 5$
- $T_E = 1$

Wastes resources!
Not a solution: local height in the graph

Need something in between
How to compute an acyclic annotation?

- Polynomial-time algorithm
  - Fix some reverse topological order on the nodes
  - For each node \( n \) in the order
    - compute the set of nodes \( S \) reachable via \((\rightarrow U \cdots)^\ast\)
    - set \( \alpha(n) = 1 + \alpha(m) \) where \( m \in S \) and \( m \) resides in the same component as \( n \)

- No unique minimal solution

```
  A   B
    ↘   ↗
    1   0

  B   A
    ↘   ↗
    0   1
```

```
  A   B
    ↘   ↗
    0   0

  B   A
    ↘   ↗
    0   0
```
OUTLINE

- Introduction
- Deadlock avoidance protocols
  - Liveness
- Distributed priority inheritance
- Related Work
- Conclusions
Basic-P: shortcomings

Basic-P may cause starvation:

A continuous stream of processes executing $G_1$ will stop processes executing $G_2$ forever
Basic-P may cause starvation:

A continuous stream of processes executing $G_1$ will stop processes executing $G_2$ forever
Liveness under Fair scheduler

Allocation Manager

{ ..... p ..... }

Scheduler

p

Allocation manager determines set of processes for which taking the resource is safe

Scheduler selects the process that gets the resource

Objective: Design protocol for allocation manager that provides liveness provided the scheduler is fair
For deadlock avoidance it is sufficient to maintain the invariant:

for every component A, for every annotation level i:

\[ t_A[i] \leq T_A - i \]

\( t_A[i] \): number of active processes in A with annotation i or greater

- Maintaining this invariant precisely requires \( T_A \) storage space

- BasicP has only one variable; it maintains the invariant by always assuming the worst case

- Question: can we maintain it precisely? What do we gain?
when \( \varphi(n) \) do

\[
\begin{align*}
\text{allocate}(n) \\
\text{f()} \\
deallocate(n)
\end{align*}
\]

when \( t_A > \alpha(n) \) do

\[
\begin{align*}
t_A++ \\
f() \\
t_A--
\end{align*}
\]

\[\begin{aligned}
\varphi(n): & \quad \forall i > \alpha(n) . t_A[i] \leq T_A - i \quad \land \quad \forall i \leq \alpha(n) . t_A[i] + 1 \leq T_A - i \\
\text{allocate}(n): & \quad \forall i \leq \alpha(n) . t_A[i]++ \\
\text{deallocate}(n): & \quad \forall i \leq \alpha(n) . t_A[i]--
\end{aligned}\]
LiveP versus BasicP

2

A

arrives

annotation level

3

2

1

0

t_A


LiveP

BasicP

Deadlock avoidance in DRE systems

Washington University, November 17, 2006
**LiveP versus BasicP**

2

A arrives

<table>
<thead>
<tr>
<th>Annotation level</th>
<th>LiveP</th>
<th>BasicP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><img src="image" alt="LiveP tA[0]" /></td>
<td><img src="image" alt="BasicP tA[0]" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image" alt="LiveP tA[1]" /></td>
<td><img src="image" alt="BasicP tA[1]" /></td>
</tr>
<tr>
<td>1</td>
<td><img src="image" alt="LiveP tA[2]" /></td>
<td><img src="image" alt="BasicP tA[2]" /></td>
</tr>
<tr>
<td>0</td>
<td><img src="image" alt="LiveP tA[3]" /></td>
<td><img src="image" alt="BasicP tA[3]" /></td>
</tr>
</tbody>
</table>

 Deadlock avoidance in DRE systems
LiveP versus BasicP

0

A arrives

annotation level

\{2\}

LiveP

\{2\}

BasicP

\[\begin{align*}
\end{align*}\]
Deadlock avoidance in DRE systems

LiveP versus BasicP

0
A
arrives

annotation level

\( t_A[0] \)
\( t_A[1] \)
\( t_A[2] \)
\( t_A[3] \)

\{0,2\}

LiveP

BasicP

Washington University, November 17, 2006
Deadlock avoidance in DRE systems

**LiveP versus BasicP**

A arrives

3

annotation level

3
2
1
0

$\{0,2\}$

LiveP

$\{0,2\}$

BasicP
**LiveP versus BasicP**

- **A** arrives

- **annotation level**
  - 3
  - 2
  - 1
  - 0

- **t_A[0]**
- **t_A[1]**
- **t_A[2]**
- **t_A[3]**

- **LiveP**
  - \(\{3, 0, 2\}\)

- **BasicP**
  - \(\{0, 2\}\)

Deadlock avoidance in DRE systems
LiveP allows more concurrency than BasicP

Theorem (Deadlock avoidance): If protocol LiveP is used to determine whether thread allocation is safe and $\alpha$ is acyclic then no deadlock is reachable.

Theorem (Liveness) If protocol LiveP is used to select a process for thread allocation then every waiting process is eventually selected.
LiveP versus BasicP

0

m2 A

arrives

1

n1 A

n2 B

0

m1 B

m2 A

annotation level

n1 n2

m1 m2

A B

A B

tA[0] tA[1]

LiveP

BasicP

tA

0

1

Deadlock avoidance in DRE systems
LiveP versus BasicP

0

\( m_2 \rightarrow A \) arrives

\[
\begin{array}{c}
\text{t}_A[0] & \text{t}_A[1] \\
\{m_2\} & \{m_2\}
\end{array}
\]

LiveP

\[
\begin{array}{c}
\text{n}_1 \rightarrow A & \text{n}_2 \rightarrow B \\
\text{m}_1 \rightarrow B & \text{m}_2 \rightarrow A
\end{array}
\]

BasicP

\[
\begin{array}{c}
\text{t}_A \\
\{m_2\}
\end{array}
\]

annotation level

1

0
LiveP versus BasicP

n1 arrives

1

\[ t_A[0] \quad t_A[1] \]

\{m2\}

LiveP

\[ \begin{array}{c}
    \text{n1 A} \\
    \text{m1 B} \\
\end{array} \quad \begin{array}{c}
    \text{n2 B} \\
    \text{m2 A} \\
\end{array} \]

annotation level

\[ \begin{array}{c}
    1 \\
    0 \\
\end{array} \quad \begin{array}{c}
    0 \\
    0 \\
\end{array} \]

\[ \begin{array}{c}
    \text{t_A[0]} \\
    \text{t_A[1]} \\
\end{array} \quad \begin{array}{c}
    \text{t_A} \\
\end{array} \]

\{m2\}
Deadlock avoidance in DRE systems

LiveP versus BasicP

LiveP

BasicP

\[
\begin{align*}
&\text{n}_1 \rightarrow A \\
&\text{t}_A[0] \quad \text{t}_A[1] \\
&\{n_1, m_2\}
\end{align*}
\]

\[
\begin{align*}
&\text{n}_1 \rightarrow A \\
&\{m_2\}
\end{align*}
\]

\[
1 \times 0
\]

\[
\begin{align*}
&\text{n}_1 \rightarrow A \\
&\text{m}_1 \rightarrow B \\
&\text{n}_2 \rightarrow B \\
&\text{m}_2 \rightarrow A
\end{align*}
\]

annotation level

\[
\begin{align*}
&\text{n}_1 \rightarrow A \\
&\text{m}_1 \rightarrow B \\
&\text{m}_2 \rightarrow A \\
&\text{n}_2 \rightarrow B
\end{align*}
\]

Deadlock avoidance in DRE systems

Washington University, November 17, 2006
LiveP versus BasicP

$m_2$ arrives

$t_{A[0]}$  $t_{A[1]}$

$\{n_1, m_2\}$

LiveP

$n_1$  $n_2$

$m_1$  $m_2$

annotation level

$t_A$

$\{m_2\}$

BasicP
LiveP versus BasicP

Deadlock avoidance in DRE systems
### LiveP versus BasicP

#### Leaves

- **LiveP**
  - $t_A[0]$  
  - $t_A[1]$  
  - $\{n_1,m_2\}$  
  - **m2 A**

- **BasicP**
  - $t_A$  
  - $\{m_2,m_2\}$  
  - **m2 A**

---

**Annotation Level**

- **LiveP**
  - $n_1 A$  

- **BasicP**
  - $m_1 B$  
  - $m_2 A$  
  - $n_2 B$
**LiveP versus BasicP**

```
0  m2  A
leaves

0
m2  A

LiveP

{n1}

{m2}

BasicP

n1  A  n2  B
m1  B  m2  A

leaves

0  0

1

m1  B  m2  A

annotation level

1  0

m2  A

{m2}

LiveP

{n1}

{m1}

BasicP

Deadlock avoidance in DRE systems
LiveP versus BasicP

m₂ A arrives

m₂ A

{m₂}

LiveP

n₁ A

{n₁}

BasicP

1

m₁ B

m₂ A

0

n₂ B

0

annotation level

{tₐ[0]} {tₐ[1]}

Deadlock avoidance in DRE systems
LiveP versus BasicP

m2 A arrives

m2 A

{n1}

LiveP

m1 B m2 A

n1 A

{n2 B}

BasicP

\{m2\} - \{m2,m2\} - \{m2\} - \{m2,m2\} ......

\{n1\} - {m2} - \{m2,m2\} - \{m2\} - {m2,m2} ......

annotation level

Deadlock avoidance in DRE systems
Basic-P vs Live-P: concurrency

Average maximum legal annotation, for $T = 10$ and varying load

Load

Maximum Legal Annotation

BasicP

LiveP

Deadlock avoidance in DRE systems
Deadlock avoidance in DRE systems

\[\text{LiveP}\]

\[\text{when } \varphi(n) \text{ do}
\]
\[
\text{allocate}(n)
\]

\[\text{when } t_A > \alpha(n) \text{ do}
\]
\[
t_A++
\]

\[\text{f()}
\]

\[\text{deallocate}(n)
\]

\[\text{f()}
\]

\[\text{t}_A--
\]

with

\[\varphi(n): \quad \forall i > \alpha(n) \cdot t_A[i] \leq T_A - i \quad \land \quad \forall i \leq \alpha(n) \cdot t_A[i] + 1 \leq T_A - i
\]

\[\text{allocate}(n): \quad \forall i \leq \alpha(n) \cdot t_A[i]++
\]

\[\text{deallocate}(n): \quad \forall i \leq \alpha(n) \cdot t_A[i]--
\]
### Live-P: implementation

- **Basic-P**: one instruction per allocation
- **Live-P**:

<table>
<thead>
<tr>
<th>data structure</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>(O(T))</td>
<td>(O(T))</td>
</tr>
<tr>
<td>binary tree</td>
<td>(O(\log T))</td>
<td>(O(T))</td>
</tr>
<tr>
<td>red-black tree</td>
<td>(O(\log L))</td>
<td>(O(L))</td>
</tr>
</tbody>
</table>

\(T\): total number of threads in the component  
\(L\): total number of processes executing in the component
Experiments (1)

Time for $10^6$ operation pairs, for uniform $\alpha$ and $L = 0$

- Array
- Complete
- RedBlack

Total number of resources $T$

Time (ms)
Experiments (2)

Time for $10^6$ operation pairs, for uniform $\alpha$ and $L = \frac{T}{4}$
LiveP versus BasicP

Deadlock avoidance in DRE systems

LiveP  BasicP

\{3,0,2\}  \{0,2\}

\(t_A[0]\)  \(t_A[1]\)  \(t_A[2]\)  \(t_A[3]\)

\(t_A\)

Annotation level:

\begin{align*}
&3 \\
&2 \\
&1 \\
&0
\end{align*}
LiveP versus BasicP

Deadlock avoidance in DRE systems
REACHABLE STATE SPACE

allocation sequences

LiveP
0 - 2 - 3
0 - 3 - 2
2 - 0 - 3
2 - 3 - 0
3 - 0 - 2
3 - 2 - 0

BasicP
3 - 2 - 0

allocation sequences

{0,2,3}

state space

LiveP
BasicP

LiveP

BasicP
Intermediate Protocols

 allocation sequences

LiveP
k-LiveP
k-LiveP for k = 2..T_A:
- k storage space
- guarantees liveness up to annotation k-1

state space
LiveP
k-LiveP
BasicP
Theorem:

Every allocation protocol that

- is enabled whenever BasicP is enabled, and
- is disabled whenever LiveP is disabled

guarantees freedom from deadlock

BasicP

LiveP
Average maximum legal annotation, for $T = 20$ and varying load

- LiveP
- EfficientP
- 5-EfficientP
- 10-EfficientP
- 15-EfficientP
- BasicP

Load vs. Maximum Legal Annotation

Deadlock avoidance in DRE systems
OUTLINE

- Introduction
- Deadlock avoidance protocols
- Liveness
- Distributed priority inheritance
- Related Work
- Conclusions
It is common in DRE systems to assign priorities to processes to ensure that more critical tasks meet their deadlines.

- Schedulers then give preference to higher-priority tasks.
- Possibility of priority inversion.
- Priority inheritance protocols help alleviate priority inversions.
- Distributed priority inheritance hard to achieve.
Priority inheritance requires a **decrease** in annotation.

Deadlock avoidance is preserved, because new state is reachable by BasicP.
OUTLINE

- Introduction
- Deadlock avoidance protocols
- Liveness
- Distributed priority inheritance
- Related Work
- Conclusions
Related work

Classical deadlock avoidance:

Flexible Manufacturing Systems:

Related work

Middleware and distributed systems:

- F. Muller, Priority inheritance and ceilings for distributed mutual exclusion, RTSS, 1999.
OUTLINE

- Introduction
- Deadlock avoidance protocols
- Liveness
- Distributed priority inheritance
- Related Work
- Conclusions
Conclusions

- BasicP ........ LiveP: a spectrum of deadlock avoidance protocols that
  - require no communication between components
  - make use of static information of the processes
  - allow varying levels of concurrency
  - provide varying levels of liveness
  - vary in complexity of implementation and space required
  - support priority inheritance

- Proofs of correctness based on parameterized transition systems

- Experimentation in ACE in progress

- Partial results for recursive call graphs
- V. Subramonian, C.D. Gill, C. Sanchez, H.B. Sipma, Reusable models for timing and liveness analysis of middleware for distributed real-time embedded systems, EMSOFT'06.
FUTURE WORK

- Formal characterization and measurement of levels of concurrency allowed by the different protocols
- Process placement to obtain optimal concurrency
- Incorporation of more information on the call graph to increase concurrency
- Static analysis methods to obtain call graphs
- Mixed WaitOnConnection - WaitOnReactor allocation policy
- Integration with scheduling
THANK YOU