# Debugging Message Passing Programs Using Invisible Message Tags


Roland Wismüller

LRR-TUM
Institut für Informatik
Technische Universtität München
Germany

e-mail: wismuell@informatik.tu-muenchen.de

# Motivation

## Current situation with parallel / distributed debugging tools:

- trace based tools (e.g. XPVM), message queue tools (e.g. MQM) show communication behavior, but no direct link to the source code
- on-line debuggers (e.g. totalview, p2d2, DETOP) allow to view execution of processes, but can't follow communication

⇒ No support for finding out how a message is processed

## Example: a client / server system

**Client**

```
send(request)
...
recv(result)
```

**Server**

```
recv(request)
STOP ....
....
send(result)
```

1. set breakpoint in server after the receipt of the request
2. step over the send call in the client

# Motivation

## This method is only possible if:

- receiver process is known (i.e. no ports, no mailboxes)
- location in the source code where request is received is known
- communication is unbuffered; no sender wildcards

$\leadsto$ in nearly all message passing environments at least one of these conditions is violated

$\Rightarrow$ **New debugger functionality is required**

## In the following:

- new commands: message breakpoints, step over communication
- implementation technique: invisible message tags
- implementation framework
- further use of invisible message tags

# Debugger Support for Message Passing

## New debugger functions:

- message breakpoint:

  *"Stop the receiver(s) of the next message sent by the specified process immediately after it received the message"*

- step over communication:

  *"Step over a send call, stop the receiver(s) of the sent message immediately after receipt"*

  ⇒ follow data flow across process boundaries

- communication level single step:

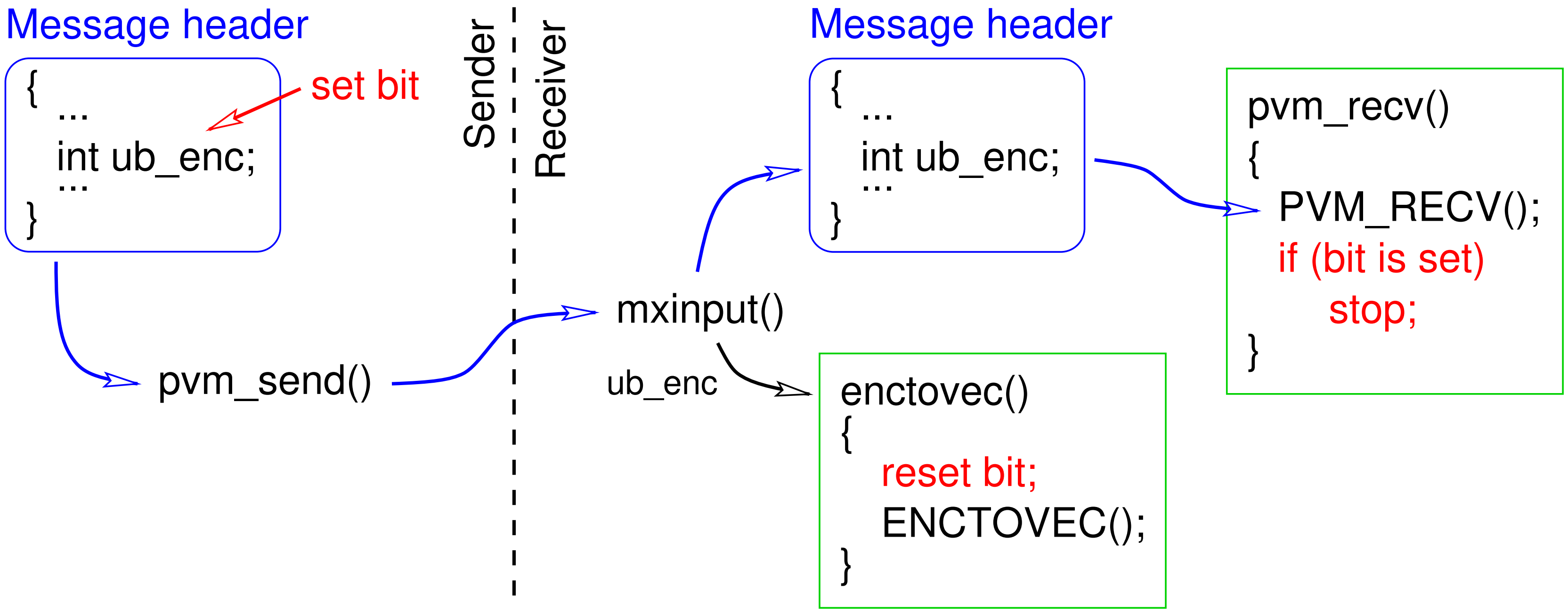
  *"Stop the specified process after it has sent the next message, and the receiver(s) of this message immediately after receipt"*

  ⇒ watch communication behavior of selected process

# Implementation Technique

## Implementation in PVM 3.3:

Use unused bits in the message header as an invisible message tag




## Other message passing libraries:

- more difficult (though not impossible) to find unused bits

$\Rightarrow$ message passing libraries should provide an additional (32-bit) field in message headers for debugging use

# Implementation Framework

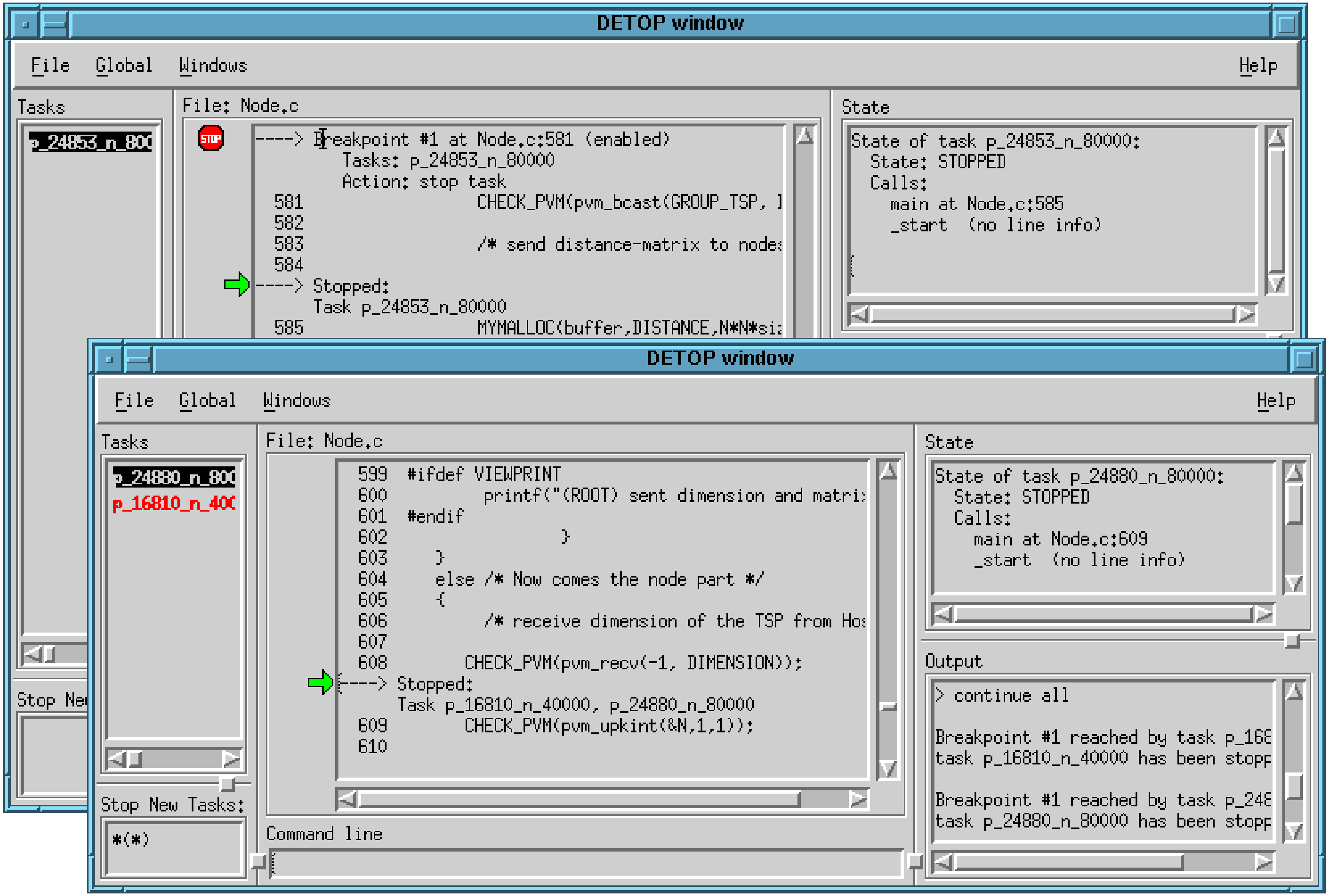
## OMIS (On-line Monitoring Interface Specification)

- universal interface for on-line monitoring of parallel and distributed programs, developed by LRR-TUM / Emory Univ.
- provides services to implement invisible message tags:
  `message_tag`: add a tag to the specified message
  `thread_has_received_tagged_msg`: tagged message has been received
  `pvm_proc_get_buffers`: get message identifiers
- see `http://wwwbode.informatik.tu-muenchen.de/~omis`

## Debugger within THE TOOL-SET for PVM

- uses an OMIS compliant monitoring system for PVM (OCM)
- allows to debug multiple processes within a single debugger window
- multiple windows make it possible to look at different codes

# THE TOOL-SET: Debugger



# Other Uses of Invisible Tags

## Distributed event detection

Answers questions such as

- *what is the first execution of procedure "foo" in process 2 that can be affected by a bug in the current line of process 1?*
- *are there any concurrent, unsynchronized writes to a file by process 1 and process 2?*

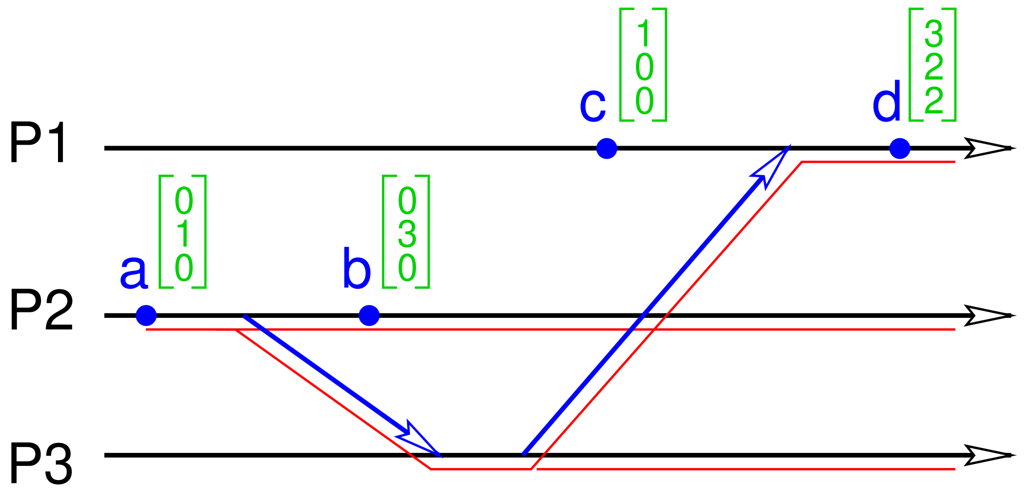
**Basis:** causality relation

$a \rightarrow b$ between events $a$ and $b$ holds, iff

- $a$ occurred prior to $b$ in the same process
- $a$ = sending of a message, $b$ = receipt of that message
- there is an event $c$ with $a \rightarrow c$ and $c \rightarrow b$

$\leadsto$ i.e. $b$ can be affected by $a$

# Distributed Event Detection




$$a \rightarrow b$$
$$a \rightarrow d$$
$$\left.\begin{array}{l} a \not\rightarrow c \\ c \not\rightarrow a \end{array}\right\} a \parallel c$$

**Traditional detection method:** virtual time: $a \rightarrow b \quad \Leftrightarrow \quad T(a) < T(b)$

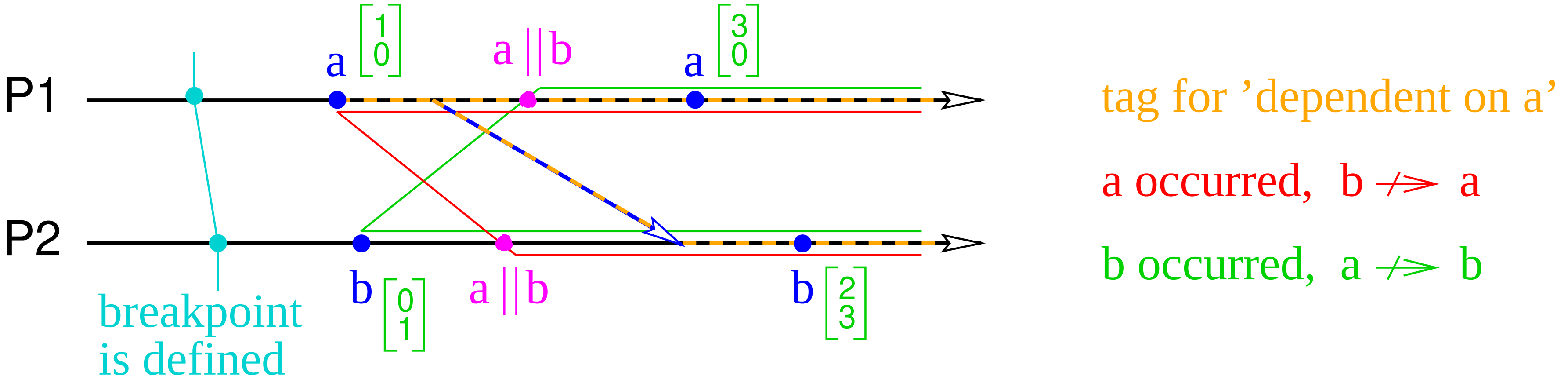
- for $n$ processes: vector of size $n$ in each process and each message
- tree-like structures are necessary for dynamic systems

## Detecting $a \rightarrow b$ using message tags

- when $a$ is detected, tag the corresponding process
- tag the messages sent by a tagged process
- tag the receivers of a tagged message

$\Rightarrow \quad a \rightarrow b$ holds, iff $b$ is detected in a tagged process

# Concurrency Predicates




## Detection method:

- two additional bits per process:
  $a$ occurred with $b \not\rightarrow a$, $b$ occurred with $a \not\rightarrow b$
- additional messages to notify event occurrences
  (required with any detection technique)
- $a \mid\mid b$ is detected when both bits are set

$\leadsto$ instantaneous detection is impossible (universal limitation)

# Conclusions, Future Work

## Conclusions

- invisible message tags enable useful additional debugger features
- they allow efficient, but limited distributed event detection
  - time stamps allow to check whether any (future) occurrences of some events $a$ and $b$ are concurrent
  - message tags only allow to check the next occurrences of $a$ and $b$

## Work in progress and work to be done

- integrate the concepts into OCM and the TOOL-SET debugger
- evaluate them in practice
- public availability: mid 1998