Cyclic Distribute d Garbage Collection Without Global Synchronization

Preliminary Examination

Gustavo Rodríguez-Rivera

Agenda

1. Area of Research
2. Importance of The Problem
3. Possible Solutions
4. Research Issues
5. Previous Work
6. Plan of Attack
7. Planned Schedule

Area of Research

Distributed System:

Multiple interconnected machines that exchange information and share resources.

Advantages:

- Improves communication
- More computing power
- More storage capacity
- Fault Tolerance

The Problem

Client-Server systems are growing in complexity
- Servers may also be clients
- Clients store state in server's space
- State may be shared by several clients
- State may need states in other servers

Problem
- When should state be deallocated?
- Clients and servers may crash while holding references to states.

Importance
- It is not possible to create complex large scale distributed systems until this problem is solved.

Possible Solutions to the Problem

1. Build stateless servers (NFS, WWW).
   + Too restrictive
   + Inefficient
   + Solves the problem only partially

2. Explicit Deallocation
   - Programmers need to agree on when to deallocate an object.
   - Leads to program errors
     Memory Leaks
     Dangling References
   - Problems of Synchronization

3. Automatic Distributed Garbage Collection
   The real solution

Concepts

Host:

Hardware + OS.

It provides multiple *Address Spaces* and *Threads*.

Process:

Address Space + one or more threads.

Objects:

Entities with state and implementation that reside in a process.

Methods:

Implementation entry points that the threads execute.

Concepts (cont.)

Distributed System:

Objects in one or more processes that interact through method invocations.

Object-id:

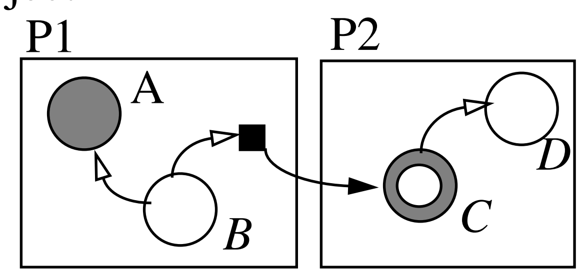
Globally unique identifier, or remote reference

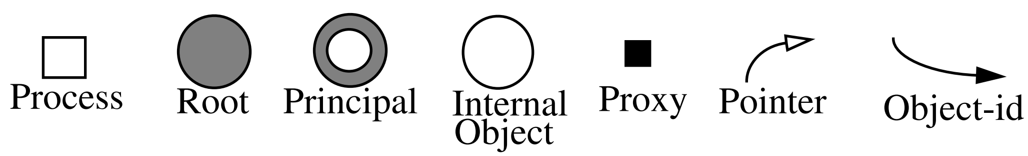
Proxy:

Local Representation of a remote object

Principal:

Exported Object





Concepts (cont.)

A directly-reaches (or *locally-reaches*) *B* if:

A is B, or

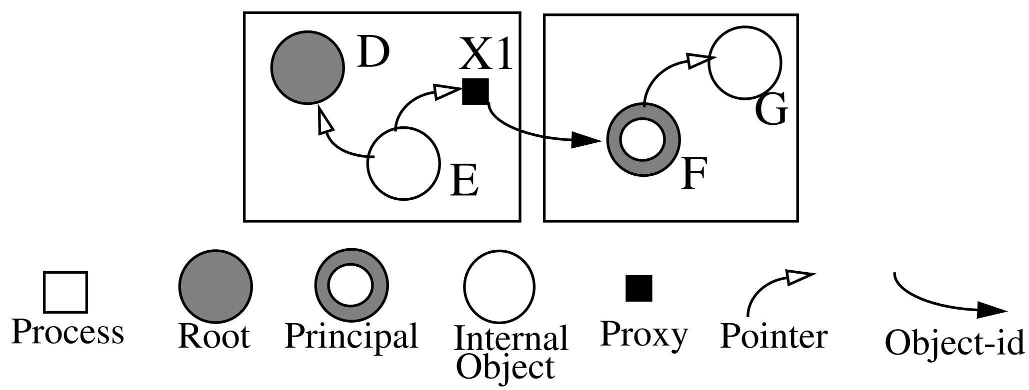
A has a pointer to *B*, or

A has a pointer to *C* and *C* directly-reaches *B*

A reaches B if:

A directly-reaches *B* or *B*'s proxy, or

A directly-reaches *C* or *C*'s proxy and *C* reaches *B*



Concepts (cont.)

Local Roots:

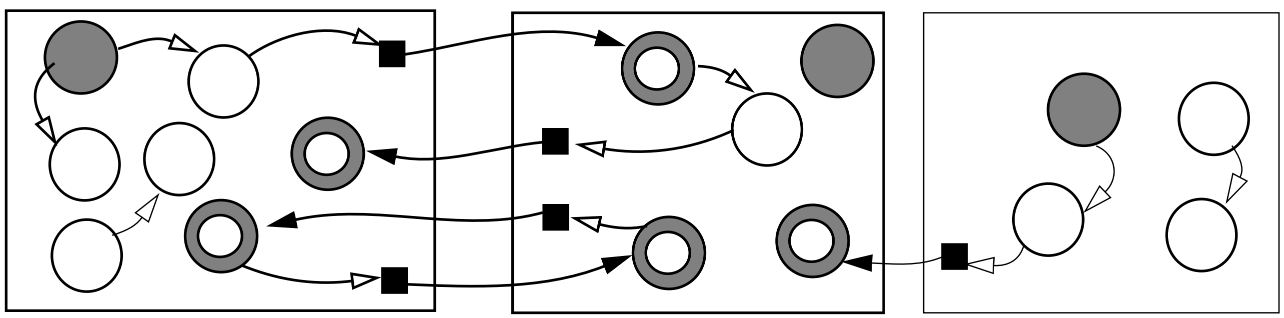
Special set of objects in a process.

Global Roots:

Union of all local roots.

Garbage Object:

Object that is not reachable from global roots..



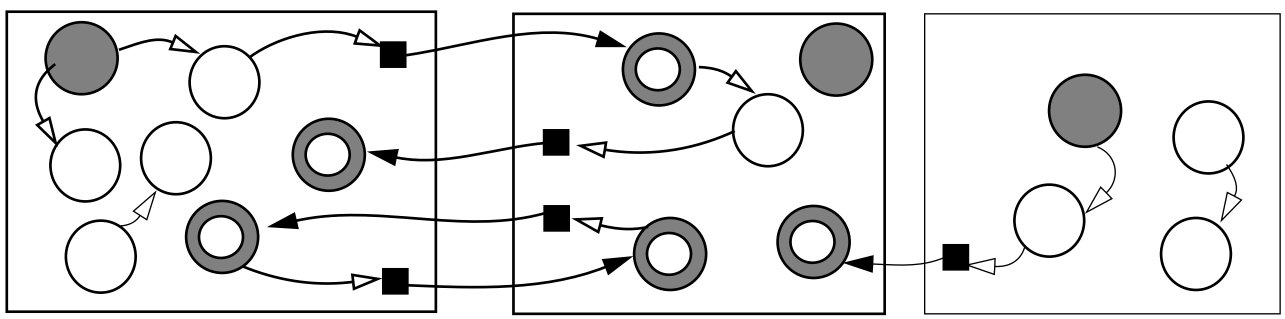
Concepts (cont.)

Local Garbage Collection:

Collects objects that are not directly-reachable from local roots or local principals.

Distributed Garbage Collection:

Collects objects that are not reachable from global roots.



Research Issues

Problems with distributed garbage collection:

- Lost and duplicated messages
- Process crashes
- Long lasting network failures
- Cycles of garbage
- Scalability
- Efficiency

There are many algorithms but none of them solves all the problems

Previous Work

Distributed Reference Counting

- Weighted Reference Counting
  [Bevan 87], [Watson & Watson 87].

- Reference Listing
  [Shapiro et al. 92], [Birrel at al. 93]

- Drawbacks:
  Does not collect cyclic garbage

Previous Work (cont.)

Tracing

- Tracing With Timestamps [Huges 85]
- Centralized Reference Service [Ladin and Liskov 92]
- Garbage Collection between Groups [Lang et al 92]

Drawbacks:

Requires global coordination (total or partial)

Hybrid

- Object Migration [Bishop 77]

Drawbacks:

Most systems do not allow migration

Need to know which objects to migrate

Plan of Attack

Given that:

Reference Listing is scalable but does not collect cycles of garbage.

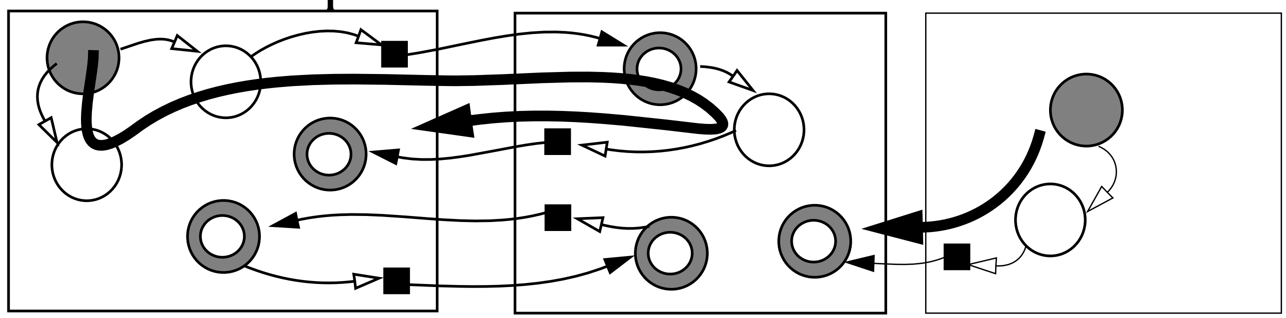
Mark & Sweep collects cycles of garbage but requires global synchronization.

None of the previous algorithms solves all the problems

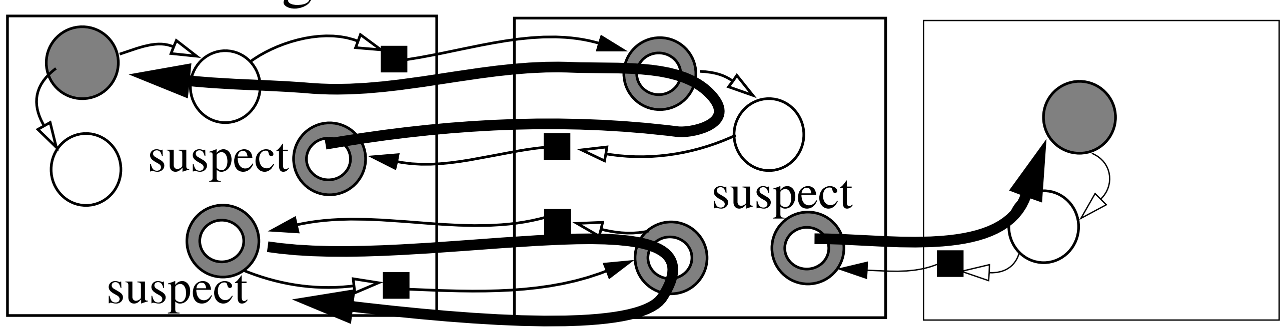
We need to design a new algorithm that collects cycles of garbage and does not require global synchronization

Back-Tracing: Basic Idea

Mark & Sweep



Back-Tracing



Back-Tracing: Problems

Need back-references to do back-tracing

- Local-back-references -> use the local garbage collector
- Remote-back-references -> keep track of proxies

How are garbage suspects determined?

- Principals not reachable from local roots are suspects

Need to synchronize multiple back-tracings

- Keep independent state for each back-tracing

Back-references may change during back-tracing

- Use sequence numbers in principals.

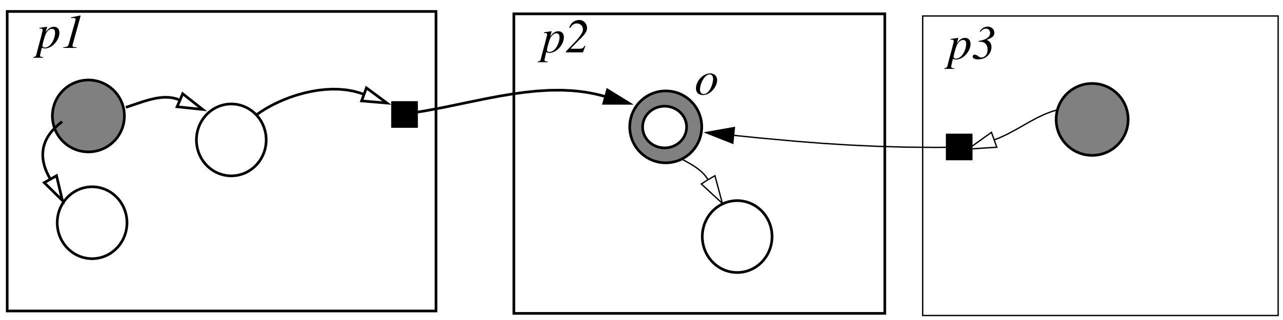
Remote-Back-References

For every principal *o* there is a variable

o.process = "processes that have a proxy to *o*"

Invariant 1:

If there is a proxy *x* for object *o* in process *p*, then *p* is in *o.process*

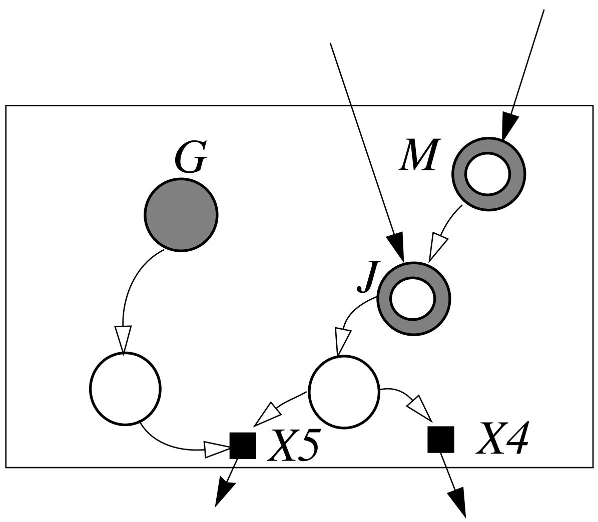


o.process = [*p1*, *p3*]

Local-Back-References

For every proxy and principal *z* there is a variable

z.roots = "principals that directly reach *z*"



J.roots = [M]
M.roots = []
X4.roots = [J, M]
X5.roots = [R, J, M]

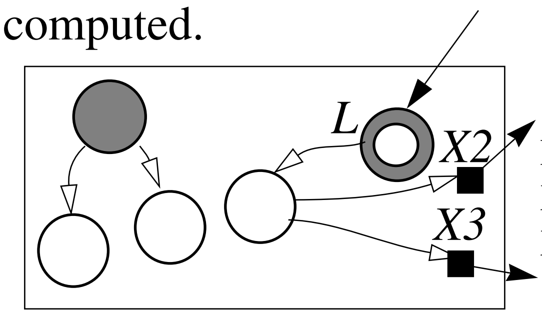
Use auxiliary tracing to compute *z.roots* during local GC.
Only proxies and principals are relevant in interprocess GC

Consistency of Back-References

For every principal *o* there is a variable *o.sequence*
that is incremented when:
- There is a remote method invocation to *o* or
- A new proxy is created for *o*

Invariant 2:

Given a proxy or principal *z*; *z.roots* is consistent if *z.roots* does not include *R* and for every principal *o* in *z.roots*, *o.sequence* has not changed since the time *z.roots* was computed.



L.sequence = 5
X2.roots = [L]
X3.roots = [L]

The Algorithm

The state of one back-tracing is stored in a single token message:

token = (principals-back-traced, proxies-to-back-trace)

At the end of a local gc, a token is created for every principal that is not reachable from a local root (*suspect*).

token = ([suspect + principals-that-directly-reach-suspect],
[all-their-proxies])

The token is passed to the owner of the first proxy *x* in *proxies-to-back-trace*. When the token is received:

x is removed from the token
Principals that reach *x* are appended to *principals-back-traced*.
Proxies that reach *x* are appended to *proxies-to-back-trace*.
This step is repeated until *proxies-to-back-trace* is empty.

The Algorithm (cont.)

If any of the principals or proxies in the token is reachable from a root, then the algorithm stops without collecting.

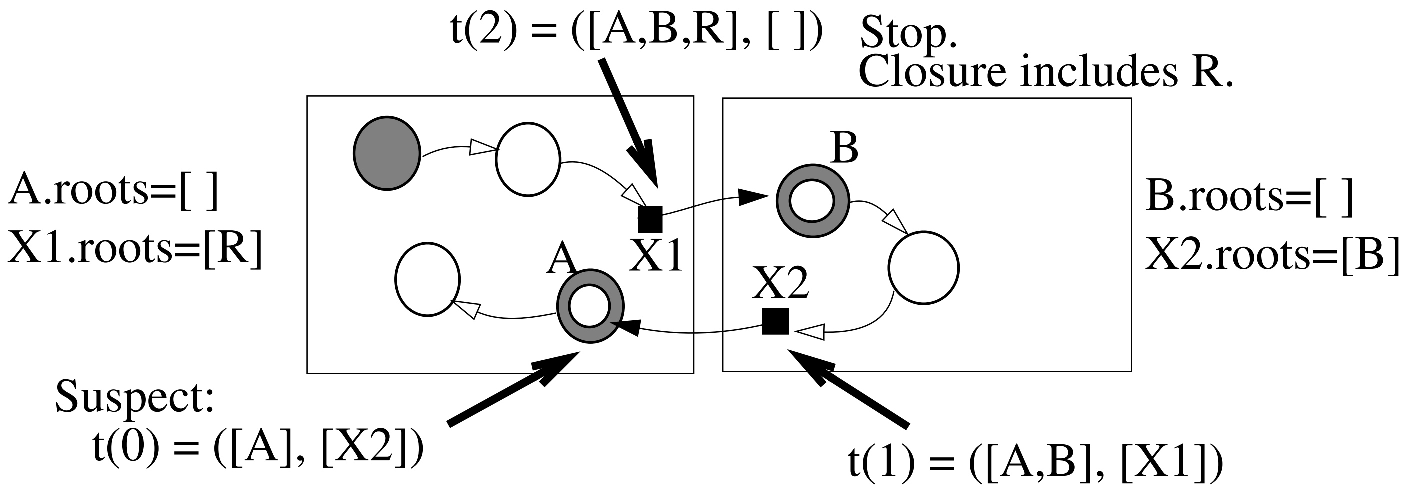
The closure is complete when there are no more proxies to back-trace.

When the closure is complete, the sequence numbers of the principals in the closure are verified.

If the sequence numbers have not changed, then the principal in the closure are reclaimed.

Example

No garbage:

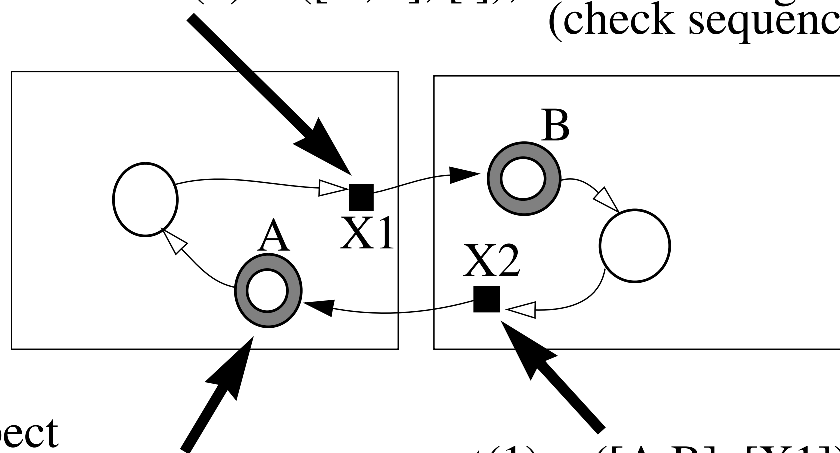


Example (cont.)

Garbage:

t(2) = ([A,B], [ ]), A and B are garbage.
(check sequence numbers first)

A.roots=[ ]
X1.roots=[A]



B.roots=[ ]
X2.roots=[B]

Suspect
t(0) = ([A], [X2])

t(1) = ([A,B], [X1])

Correctness and Completeness

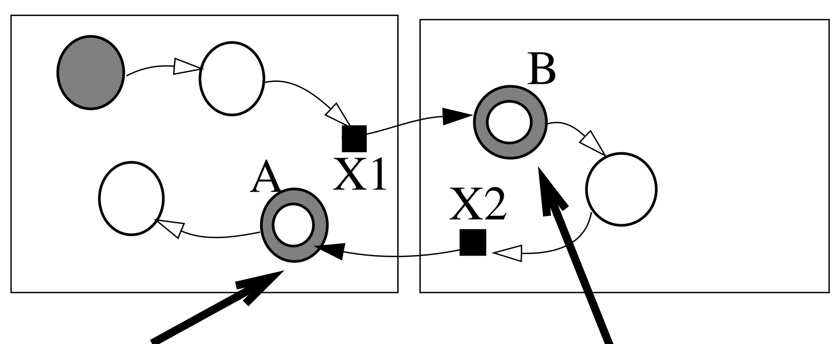
Correctness: Objects collected are garbage

Completeness: Garbage objects will sooner or later be collected.

Invariant 3: Let token t = *(principals, proxies)*; if the sequence numbers of the objects in *principals* have not changed and there is a root that reaches an object in principals, then the root reaches the object through a proxy in *proxies*.

Proof by induction in the number of processes.

A.roots=[ ]
X1.roots=[R]



B.roots=[ ]
X2.roots=[B]

Suspect:
t(0) = ([A], [X2])

t(1) = ([A,B], [X1])

Correctness and Completeness (cont.)

Corollary to Invariant 3:

Let token t = *(principals, proxies)*; if the sequence numbers of the objects in *principals* have not changed and *proxies* is empty, then no root reaches an object in principals.

Proof by contradiction.

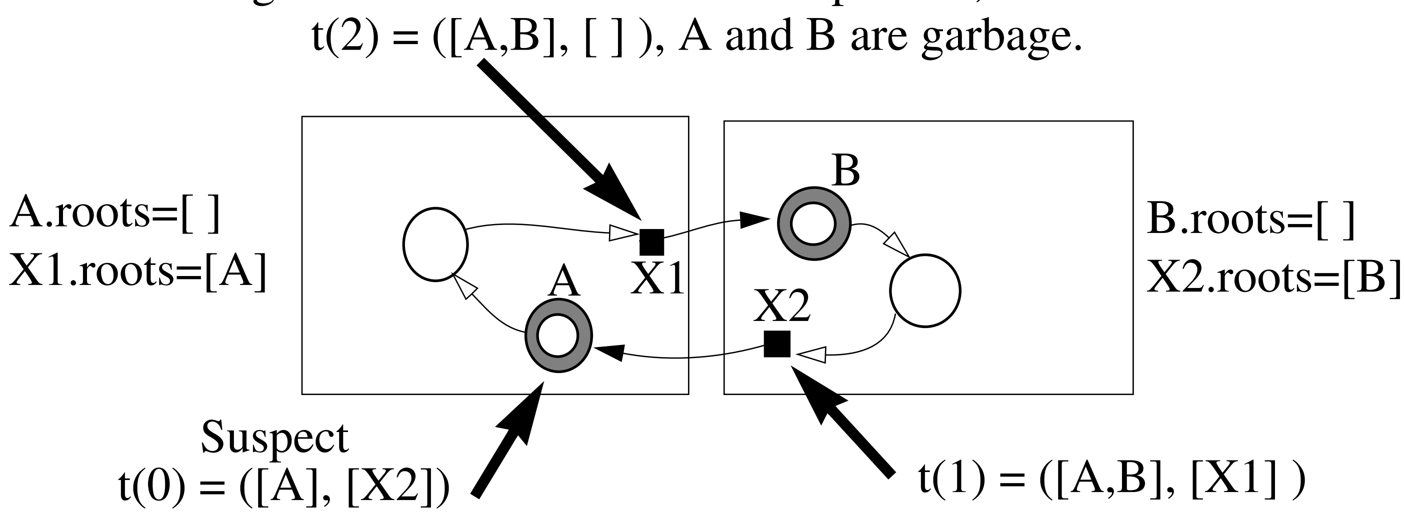
Proof of correctness:

Use Corollary to Invariant 3.

Correctness and Completeness (cont.)

Proof of completeness:

- Assume local garbage collections run periodically
- For every interprocess garbage graph there is a “suspect” principal that is not reachable from a local root from which a back-tracing starts.
- Every time a token visits a process, one proxy is removed.
- Proxies are added to a token only once.
- Garbage have a bounded number of proxies, so collection will end.



Fault Tolerance

Lost and Duplicated Messages

- If a token is lost, another token is generated during the next local GC
- If a token is duplicated, one will succeed and the others will not.

Process crash:

- Well known server in each host that gives process status.
- If pass-token operation times out, ask for status of recipient process.
- If process has crashed, discard token. Remove recipient process from all local principals.
- If recipient process has not crashed or server does not respond, then discard token and try later.

Unsolved Issues

- So far I have designed the algorithm and proved its correctness.
- Will it work? Need to implement it.
- Is it efficient? Do performance analysis
- Tokens message may be too large. Try other solutions.
- Other optimizations:
    - Minimize number of suspects. Less back-tracings
    - Minimize space used.
- Still a lot to do

Planned Schedule

Fall-95, Spring-96:

Implement algorithm in Renaissance
Get performance analysis
Optimize algorithm
Publish papers.

Fall-96, Spring-97

Write thesis. And final defense.