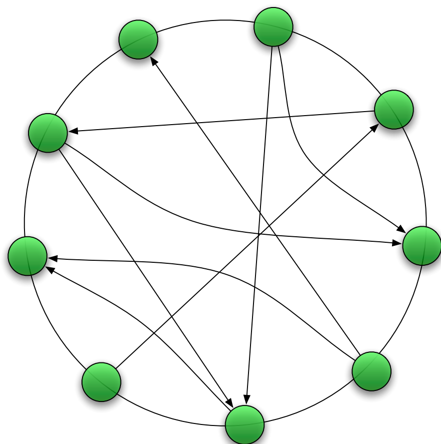
P2PS: Distributed Transaction on top of the Relaxed-Ring

Boriss Mejías and the distoz group

Université catholique de Louvain,
Louvain-la-Neuve, Belgium
boriss.mejias@uclouvain.be

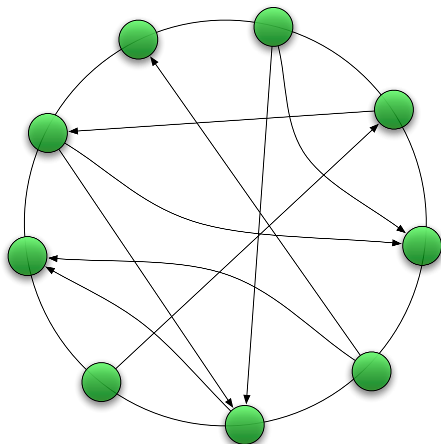
21st Nov, 2008

Chord peer-to-peer Network



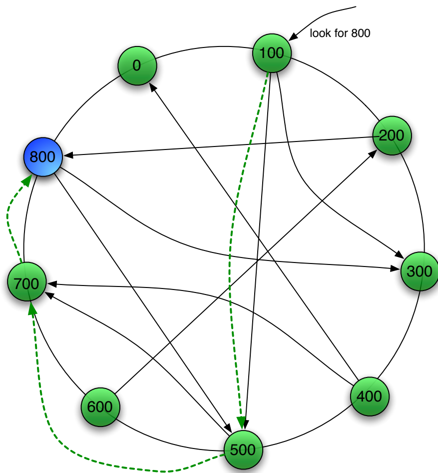
- Scalable
- Fully decentralized
- Self-organized

Chord peer-to-peer Network



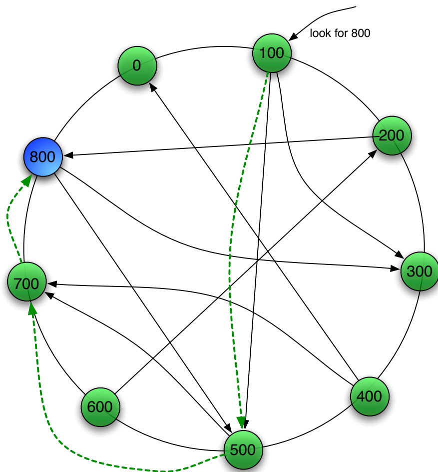
- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$

Chord peer-to-peer Network



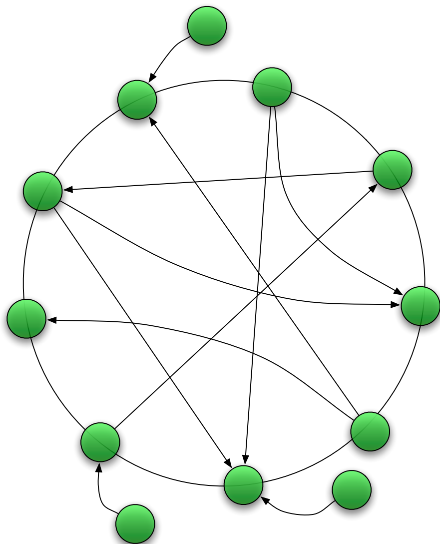
- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$

Chord peer-to-peer Network



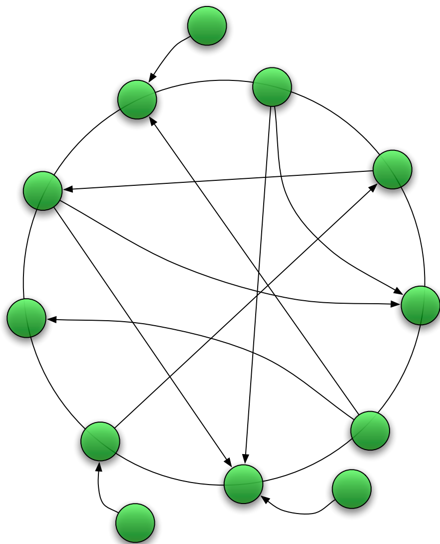
- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N))$
- Lookup inconsistencies (due to churn)
- Expensive maintenance (periodic stabilization)

Relaxed-Ring peer-to-peer Network



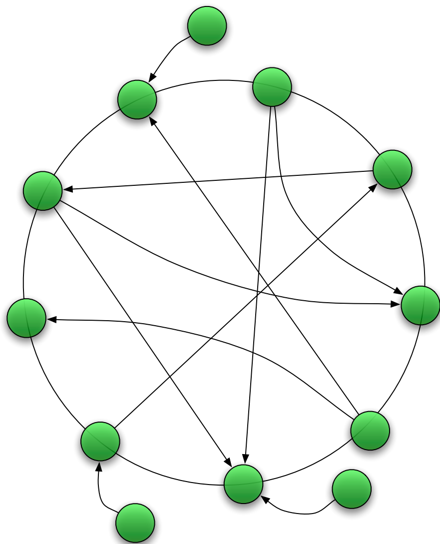
- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N) + b)$

Relaxed-Ring peer-to-peer Network



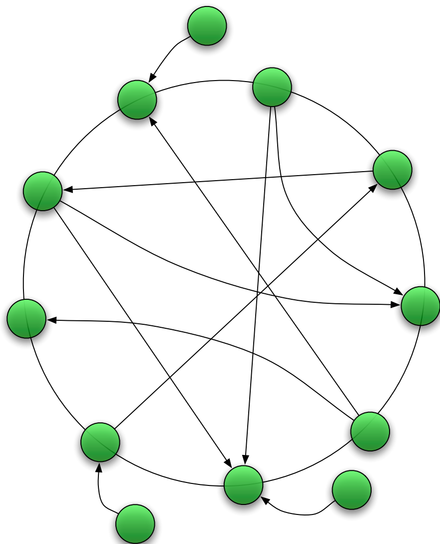
- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N) + b)$
- Almost no lookup inconsistencies

Relaxed-Ring peer-to-peer Network



- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N) + b)$
- Almost no lookup inconsistencies
- Join algorithm in two steps involving two nodes (instead of one step involving 3 nodes)
- Cost-efficient maintenance

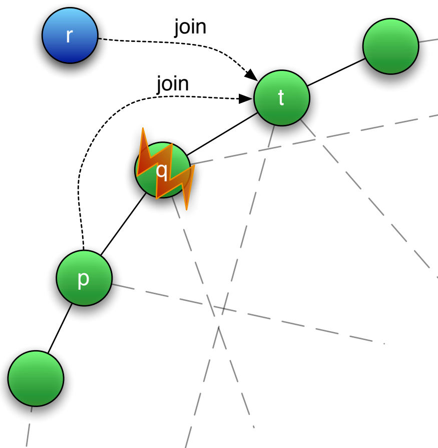
Relaxed-Ring peer-to-peer Network



- Scalable
- Fully decentralized
- Self-organized
- Efficient routing $O(\log(N) + b)$
- Almost no lookup inconsistencies
- Join algorithm in two steps involving two nodes (instead of one step involving 3 nodes)
- Cost-efficient maintenance
- It handles peers with limited connectivity

Failure Recovery

- Peer *p* and *t* detects the crash of peer *q*
- Only *q*'s predecessor (*p*) triggers recovery
- Recovery message is equivalent to *join* message of a new peer
- It does not matter the order of *join* messages sent by *p* and *r* arriving at peer *t*



P2PS

- P2PS implements the Relaxed-Ring
- It provides a simple DHT on top of it
- It provides distributed transactions with symmetric replication
- It can run as simulation using light-weight threads
- It can run as a real network using the distribution layer of Oz 3 or the integration with Dss in Oz 4
- It is organized using Tiers
- Each tier provides a different functionality: session, relaxed-ring, routing, watcher, dht, transaction

P2PS API

- Creating a Node

```
import
    P2PSNode at «P2PSNode.ozf«
define
    Node = {P2PSNode.newP2PSNode Args}
```

- Args:

```
args(maxKey:<INT>                    % 100000
     succListSize:<INT>              % 4
     dist:<sim,dl,dss>               % sim
     branch:<INT>                    % no branch
     routing:<chord,dks...>          % tango
     k:<INT>                         % 13
     transactions:<BOOL>             % false
    )
```

P2PS API

- Joining a network

```
RingRef = {N1 getRingRef($)}
{N2 join(RingRef)}

Succ = {Node getSuccRef($)}
Id = {Node getId($)}
{Node leave}
{Node injectPermFail}
```

P2PS API

- Joining a network

```
RingRef = {N1 getRingRef($)}
{N2 join(RingRef)}

Succ = {Node getSuccRef($)}
Id = {Node getId($)}
{Node leave}
{Node injectPermFail}
```

- Send messages

```
{Node sendTo(Id Msg)}
{Node sendTo(Id Msg responsible:false)}
{Node rSendTo(Id Msg delivered:Flag)}
{Node broadcast(Range Msg)} % all, butMe, From#To
```

P2PS API

- Joining a network

```
RingRef = {N1 getRingRef($)}
{N2 join(RingRef)}

Succ = {Node getSuccRef($)}
Id = {Node getId($)}
{Node leave}
{Node injectPermFail}
```

- Send messages

```
{Node sendTo(Id Msg)}
{Node sendTo(Id Msg responsible:false)}
{Node rSendTo(Id Msg delivered:Flag)}
{Node broadcast(Range Msg)} % all, butMe, From#To
```

- DHT

```
{Node put(Key Value)}
{Node get(Key Value)}
```

P2PS API - Transactions

- Transactions

```
{Node executeTransaction(F Client Protocol)}
% Client: <Port>
% Protocol: twopc, paxos

F = proc {$ Obj}
        {Obj write(Key Val)}
        {Obj read(Key Val)}
        if Value == 42 then
           {Obj abort}
        else
           {Obj commit}
        end
     end
```

Examples on Transactions

- How to write two different items using Paxos Consensus Algorithm

```
declare
Stream Client
Trans = proc {$ Obj}
            {Obj write(hello "Charlotte")}
            {Obj write(foo bar)}
            {Obj commit}
        end
{NewPort Stream Client}
{Node executeTransaction(Trans Client paxos)}
if Stream.1 == commit then
   {Browse "transaction succeeded"}
end
```

Examples on Transactions

- How to read the values we just inserted in the distributed database

```
declare
V1 V2
Trans2 = proc {$ Obj}
            {Obj read(hello V1)}
            {Obj read(foo V2)}
         end
{Node executeTransaction(Trans2 Client paxos)}
{Browse "for hello I got"#V1}
{Browse "for foo I got"#V2}
```