Mobile Crowd Computing & Task Farming

Jon Crowcroft&Eiko Yoneki

Jon.crowcroft@cl.cam.ac.uk

Two Part Talk

- First, talk about Mobile Cloud Computing Programming Models
- Second, talk about task farming in MCC, and encounter statistics impact on performance

Part 1 -
Programming Distributed Computation in Pocket Switched Networks (CCN/NDN etc)

came out of random (good) question by Brad Karp during Pan Hui's PhD defense

* **D**ata **D**riven **D**eclarative **N**etworking

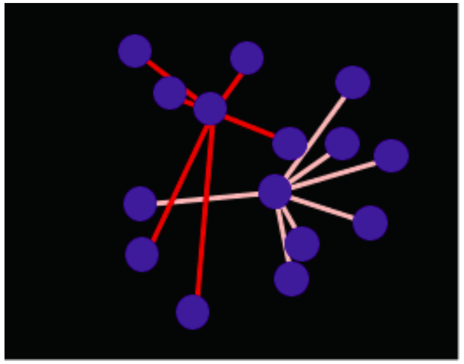
PSN: Dynamic Human Networks

- Topology changes every time unit
- Exhibits characteristics of Social Networks

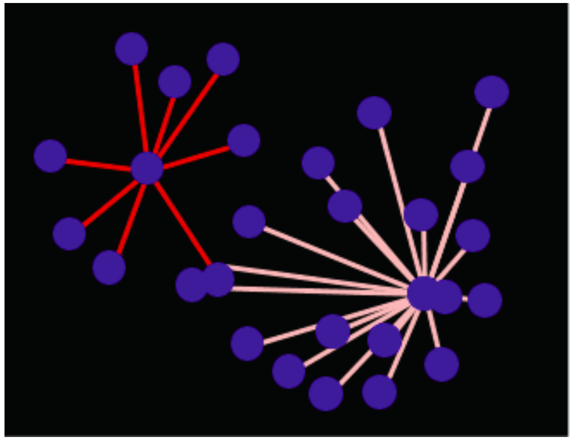
● Node

— High weight edge

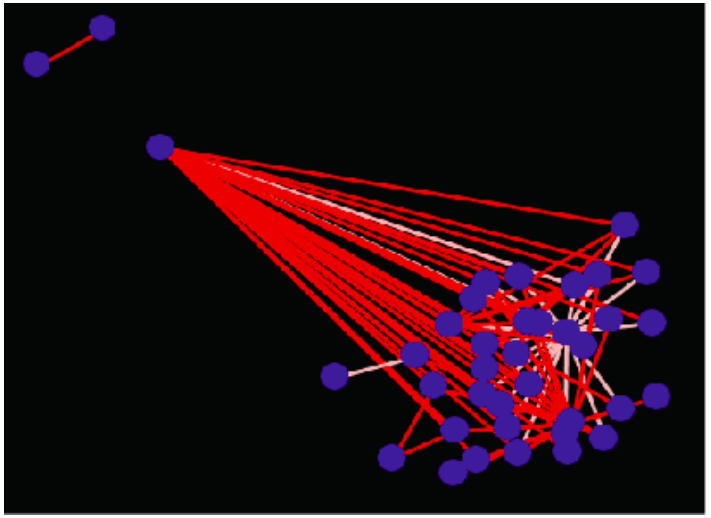
— Low weight edge



Time unit = t



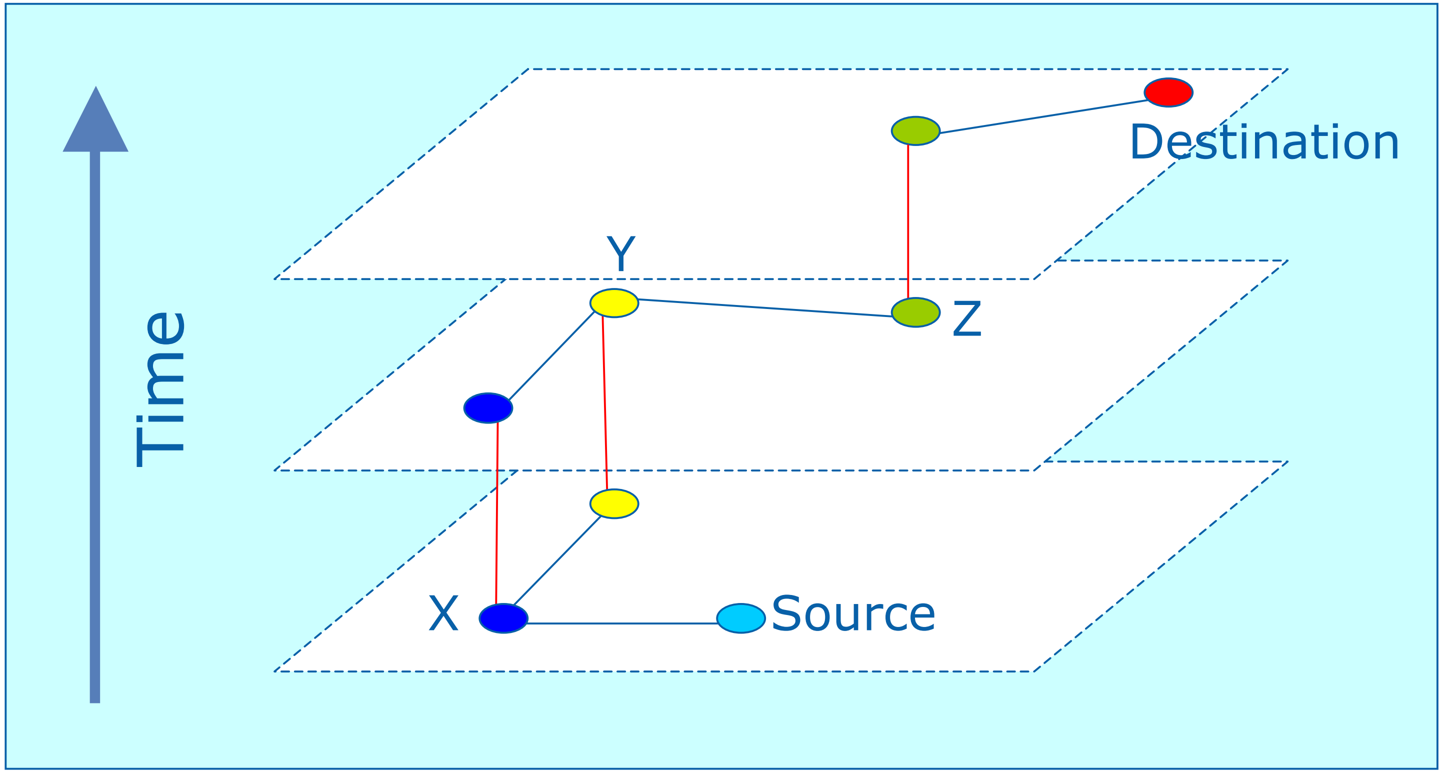
Time unit = t+1



Time unit = t+2

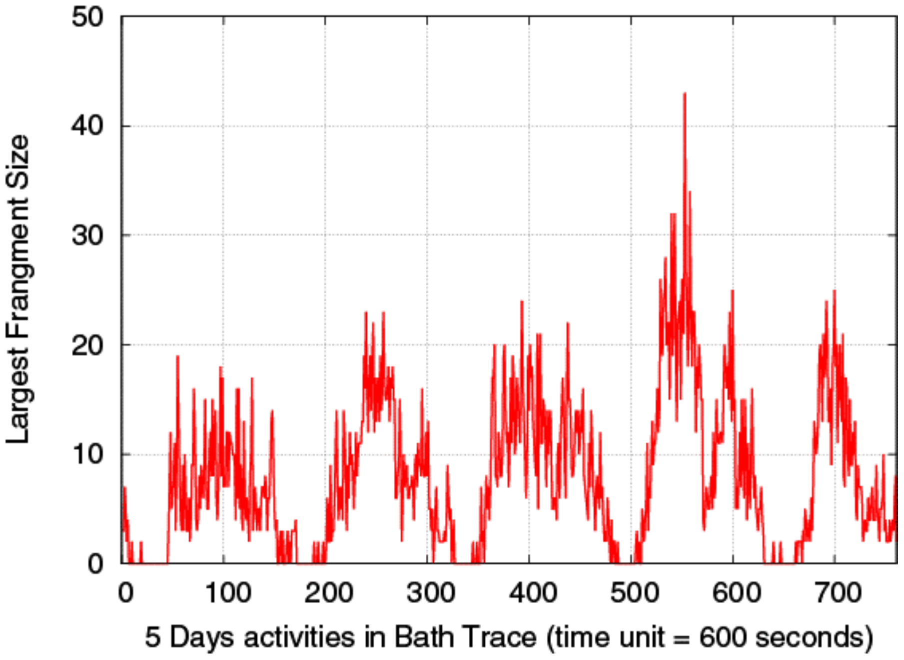
Time Dependent Networks

- Data paths may not exist at any one point in time but do exist *over time*
- Delay Tolerant Communication

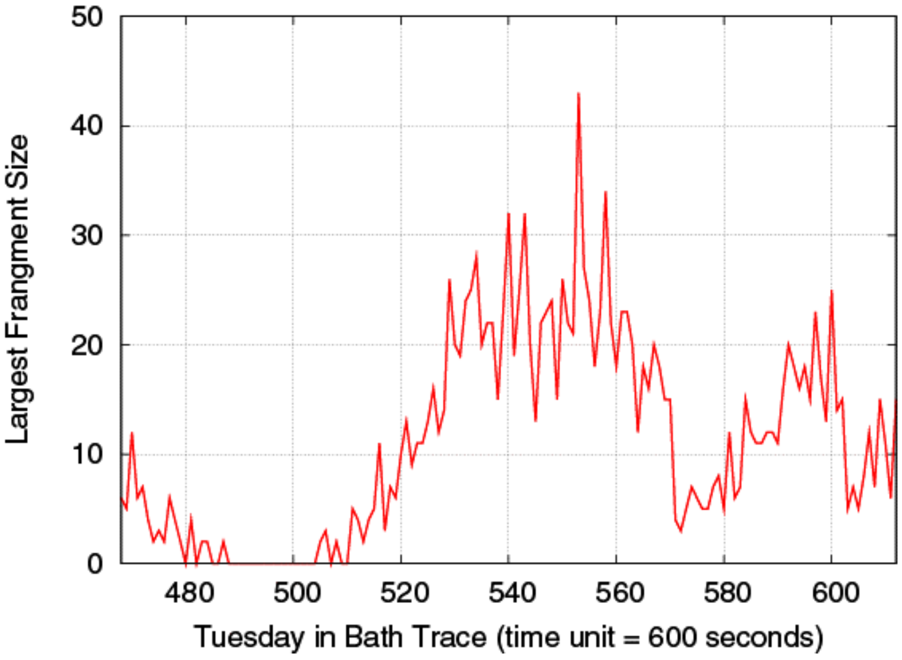


Regularity of Network Activity

- Size of largest fragment shows network dynamics



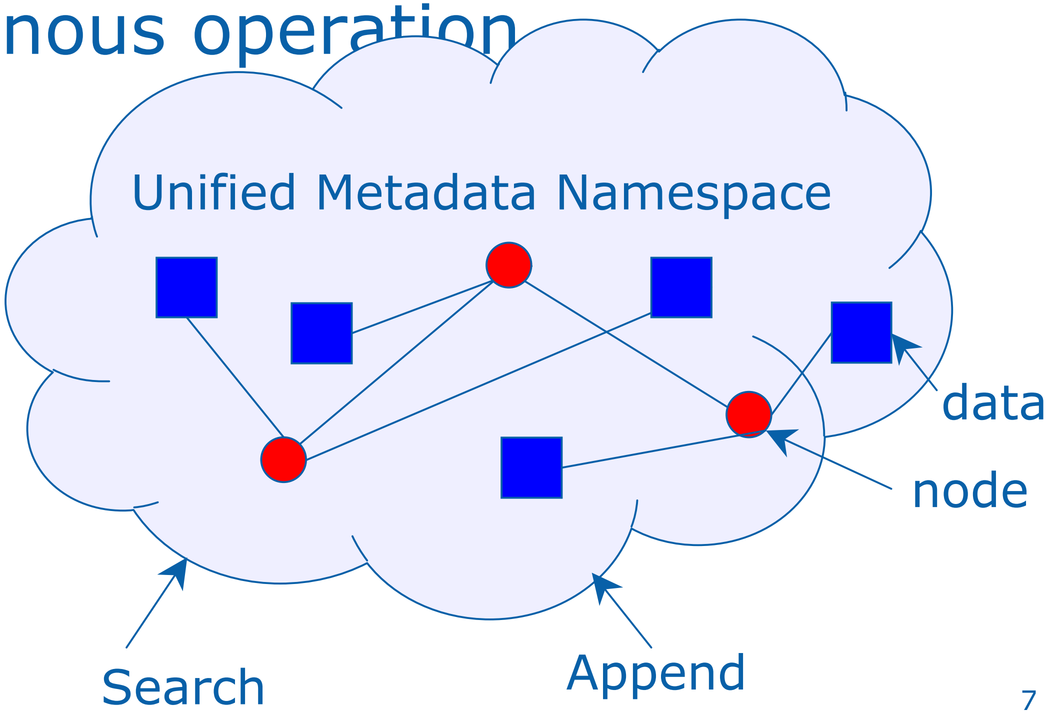
5 Days



Tuesday

Haggle Node Architecture = Runtime

- Each node maintains a data store: its current view of global namespace
  - Persistence of search: delay tolerance and opportunism
- Semantics of publish/subscribe and an event-driven + asynchronous operation
- Multi-platform
  (written in C++ and C)
  - Windows mobile
  - Mac OS X, iPhone
  - Linux
  - Android



D^3N Data-Driven Declarative Networking

- How to program distributed computation?
  - Use Declarative Networking ?

- The Vodafone Story….
  - Need tested or verified code….so also good…
  - Three reasons:

1. No PII leakage
2. No crashes
3. No unexplained bills….

Declarative Networking

- Declarative is not now a very new idea in networking
    - e.g. Search: ‘what to look for’ rather than ‘how to look for’
    - Abstract complexity in networking/data processing
- **P2**: Building overlay using Overlog
    - Network properties specified declaratively
- **LINQ**: extend .NET with language integrated operations for query/store/transform data
- **DryadLINQ**: extends LINQ similar to Google’s Map-Reduce
    - Automatic parallelization from sequential declarative code

- **Opis**: Functional-reactive approach in OCaml

D³N Data-Driven Declarative Networking

- How to program distributed computation?

- Use Declarative Networking
  - Use of Functional Programming
    - Simple/clean semantics, expressive, inherent parallelism
  - Queries/Filer etc. can be expressed as higher-order functions that are applied in a distributed setting
- Runtime system provides the necessary native library functions that are specific to each device
  - Prototype: F# + .NET for mobile devices

D^3N and Functional Programming I

- Functions are first-class values
    - They can be both input and output of other functions
    - They can be shared between different nodes (code mobility)
    - Not only data but also functions flow
- Language syntax does not have state
    - Variables are only ever assigned once; hence reasoning about programs becomes easier

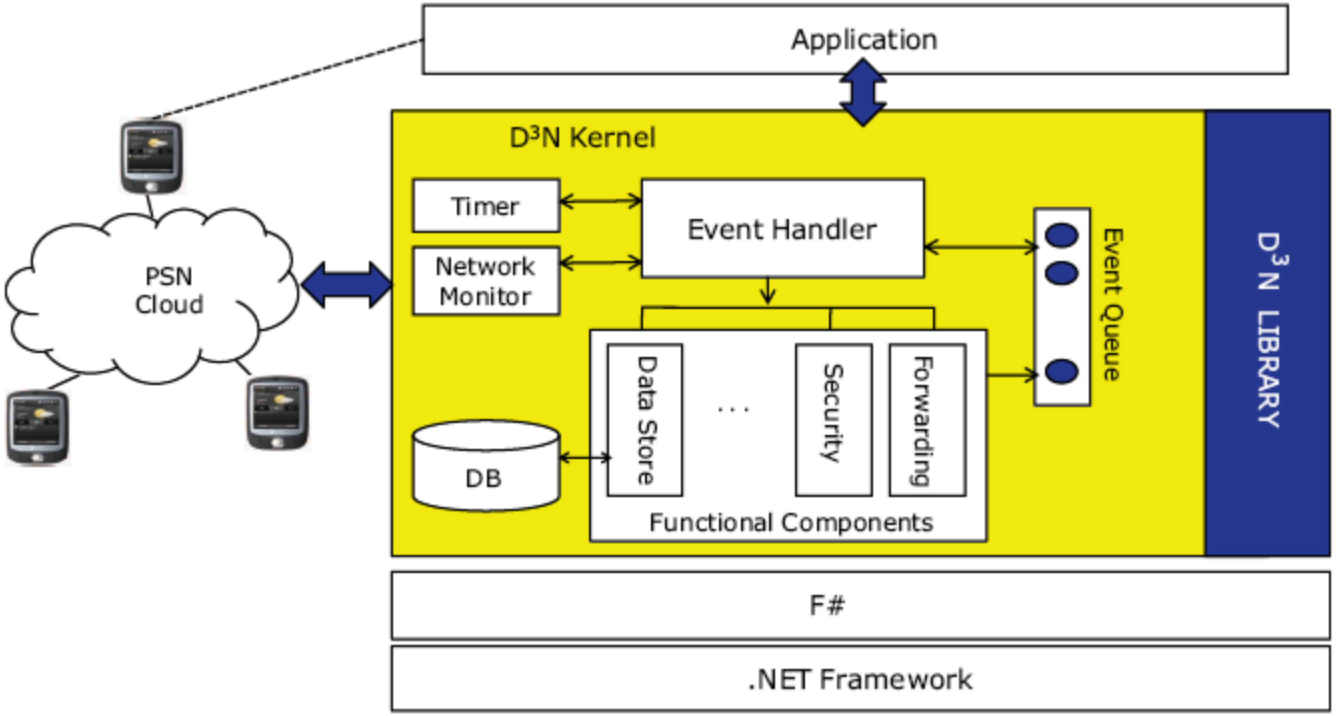
  (of course message passing and threads → encode states)
- Strongly typed
    - Static assurance that the program does not ‘go wrong’ at runtime unlike script languages
- Type inference
    - Types are not declared explicitly, hence programs are less verbose

D^3N and Functional Programming II

- Integrated features from query language
    - Assurance as in logical programming
- Appropriate level of abstraction
    - Imperative languages closely specify the implementation details (how); declarative languages abstract too much (what)
    - Imperative – predictable result about performance
    - Declarative language – abstract away many implementation issues

Overview of D^{3}N Architecture

- Each node is responsible for storing, indexing, searching, and delivering data
- Primitive functions associated with core D^{3}N calculus syntax are part of the runtime system
- **Prototype on MS Mobile .NET**



D^{3}N Syntax and Semantics I

- Very few primitives
  - Integer, strings, lists, floating point numbers and other primitives are recovered through constructor application

- Standard FP features
  - Declaring and naming functions through let-bindings
  - Calling primitive and user-defined functions (function application)
  - Pattern matching (similar to switch statement)
  - Standard features as ordinary programming languages (e.g. ML or Haskell)

D^3N Syntax and Semantics II

- Advanced features
  - Concurrency (fork)
  - Communication (send/receive primitives)
  - Query expressions (local and distributed select)

Runtime System

- Language relies on a small runtime system
    - Operations implemented in the runtime system written in F#
- Each node is responsible on data:
    - Storing, Indexing, Searching
    - Delivering
    - Data has Time-To-Live (TTL)
    - Each node propagates data to the other nodes.
    - A search query w/TTL travels within the network until it expires
    - When the node has the matching data, it forwards the data
    - Each node gossips its own metadata when it meets other nodes

Example: Query to Networks

- Queries are part of source level syntax
    - Distributed execution (single node programmer model)
    - Familiar syntax

D^{3}N:

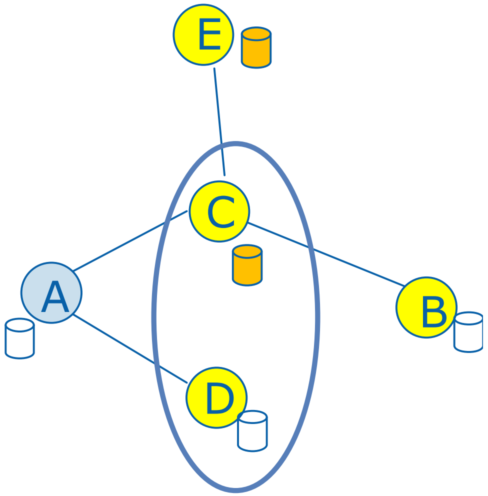
```
select name from poll() where institute = “Computer Laboratory”
```

F#:

```
poll()
|> filter (fun r -> r.institute = “Computer Laboratory”)
|> map (fun r -> r.name)
```

Message:

```
(code, nodeid, TTL, data)
```



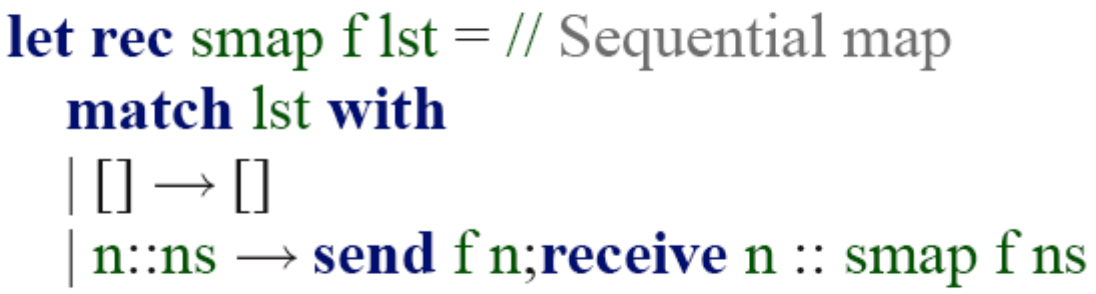
Example: Vote among Nodes

- Voting application: implements a distributed voting protocol of choosing location for dinner

- Rules
  - Each node votes once
  - A single node initiates the application
  - Ballots should not be counted twice
  - No infrastructure-base communication is available or it is too expensive
- Top-level expression
  - Node A sends the code to all nodes
  - Nodes map in parallel (pmap) the function voteOfNode to their local data, and send back the result to A
  - Node A aggregates (reduce) the results from all nodes and produces a final tally

Sequential Map function (smap)

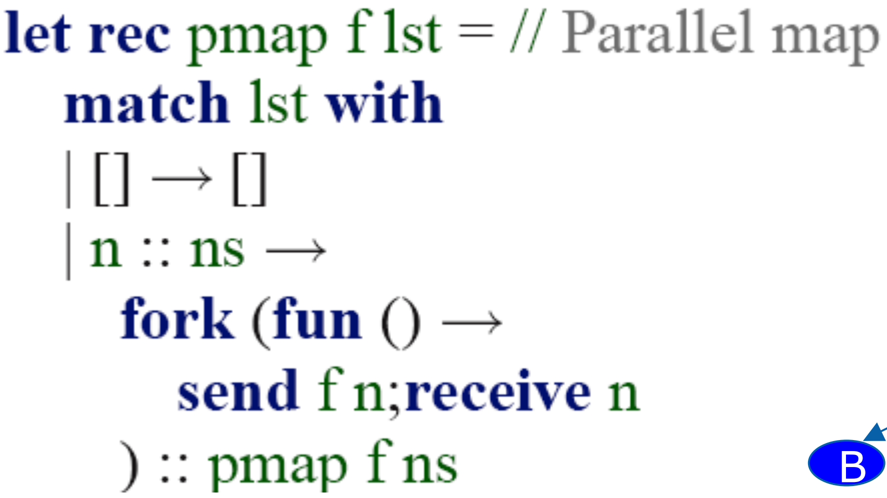
- Inner working
    - It sends the code to execute on the remote node
    - It blocks waiting for a response waiting from the node
    - Continues mapping the function to the rest of the nodes in a sequential fashion
    - An unavailable node blocks the entire computation



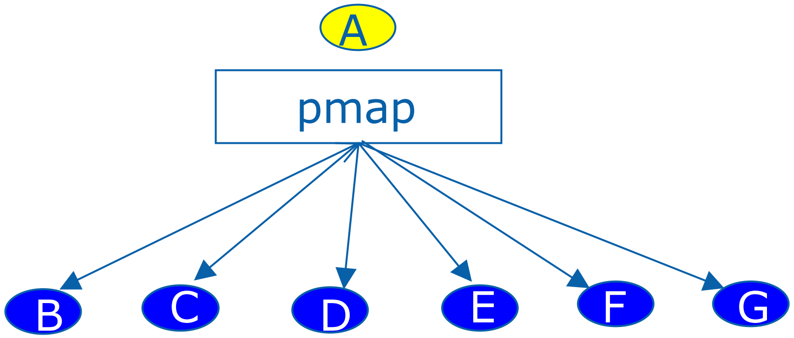
```
let rec smap f lst = // Sequential map
    match lst with
    | [] → []
    | n::ns → send f n;receive n :: smap f ns
```

Parallel Map Function (pmap)

- Inner working
  - Similar to the sequential case
  - The send/receive for each node happen in a separate thread
  - An unavailable node does not block the entire computation



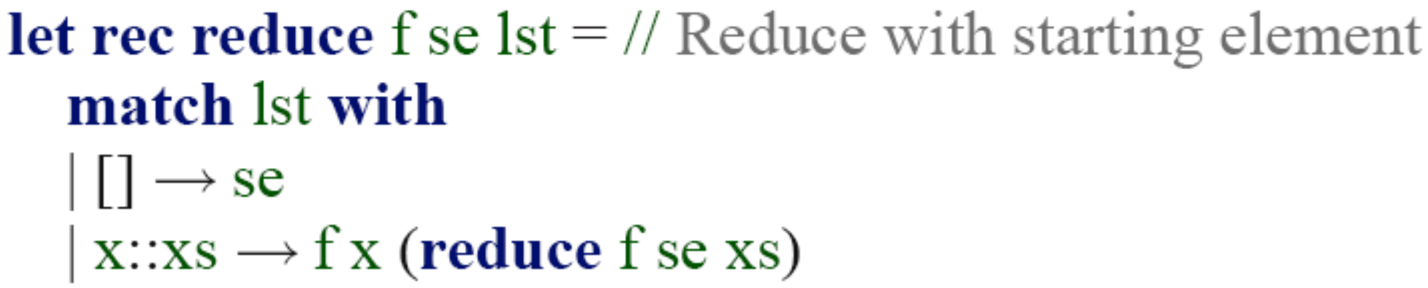
```
let rec pmap f lst = // Parallel map
   match lst with
   | [] → []
   | n :: ns →
      fork (fun () →
         send f n;receive n
      ) :: pmap f ns
```



```
Event.register( Event.OnEncounter, fun d:device ->
  if d.nID = “B” && distance(self,d) < 3 then
    dispatch NodeEncountered(d);
)
```

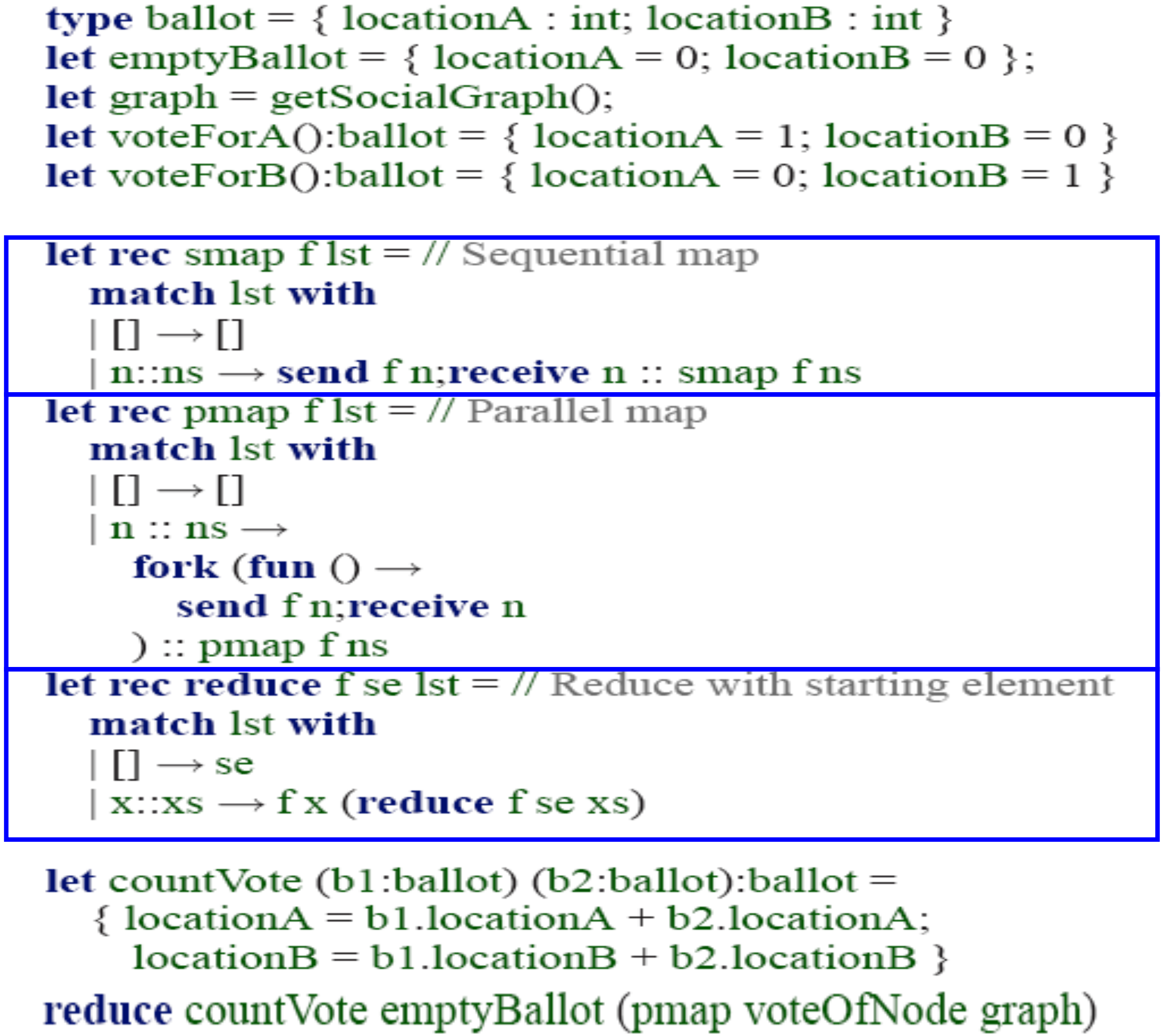
Reduce Function

- Inner working
  - The reduce function aggregates the results from a map
  - The reduce gets executed on the initiator node
  - All results must have been received before the reduce can proceed



```
let rec reduce f se lst = // Reduce with starting element
    match lst with
    | [] → se
    | x::xs → f x (reduce f se xs)
```

Voting Application Code

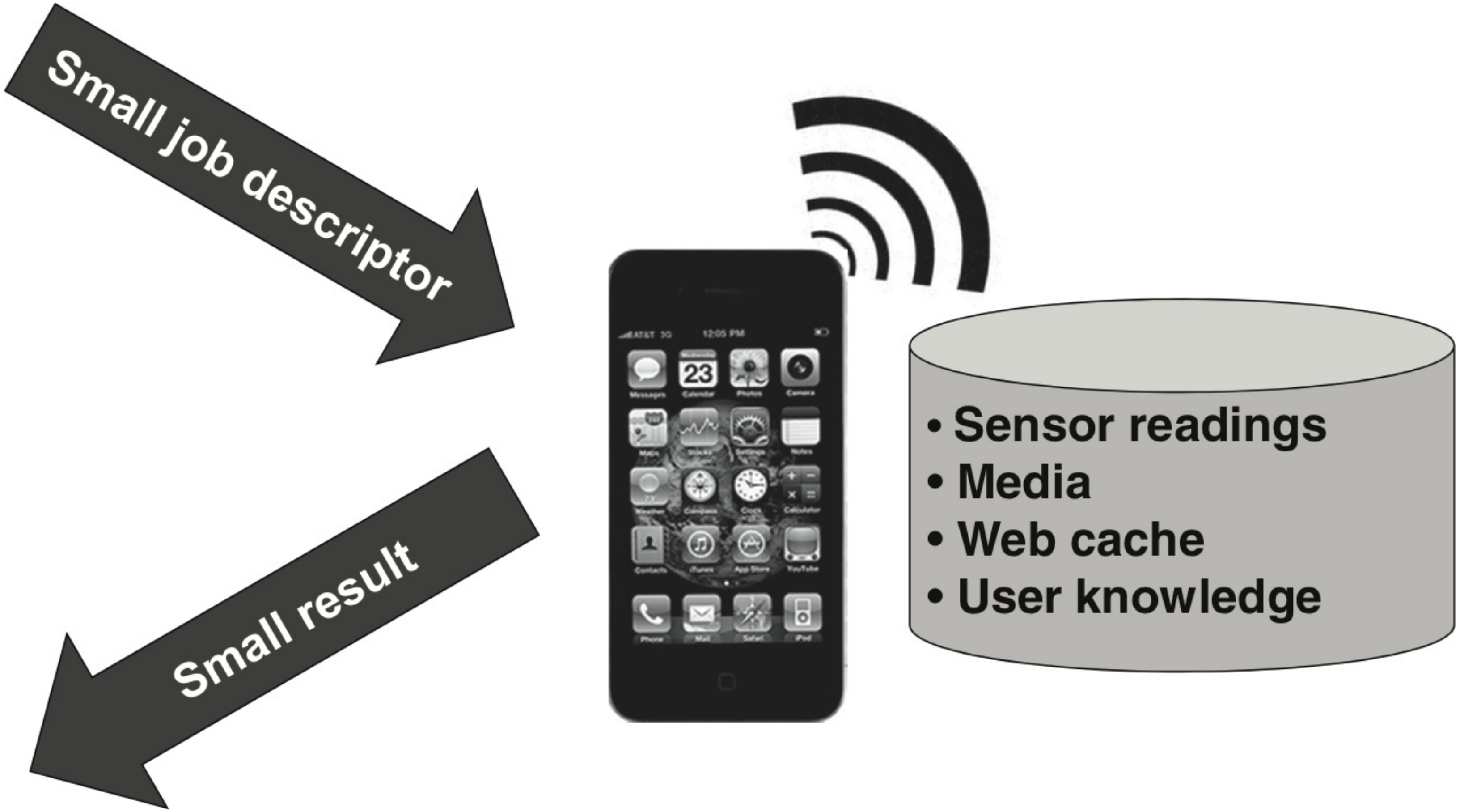


```
type ballot = { locationA : int; locationB : int }
let emptyBallot = { locationA = 0; locationB = 0 };
let graph = getSocialGraph();
let voteForA():ballot = { locationA = 1; locationB = 0 }
let voteForB():ballot = { locationA = 0; locationB = 1 }

let rec smap f lst = // Sequential map
   match lst with
   | [] ⟶ []
   | n::ns ⟶ send f n;receive n :: smap f ns
let rec pmap f lst = // Parallel map
   match lst with
   | [] ⟶ []
   | n :: ns ⟶
      fork (fun () ⟶
         send f n;receive n
      ) :: pmap f ns
let rec reduce f se lst = // Reduce with starting element
   match lst with
   | [] ⟶ se
   | x::xs ⟶ f x (reduce f se xs)

let countVote (b1:ballot) (b2:ballot):ballot =
   { locationA = b1.locationA + b2.locationA;
      locationB = b1.locationB + b2.locationB }
reduce countVote emptyBallot (pmap voteOfNode graph)
```

Outlook and Future Work

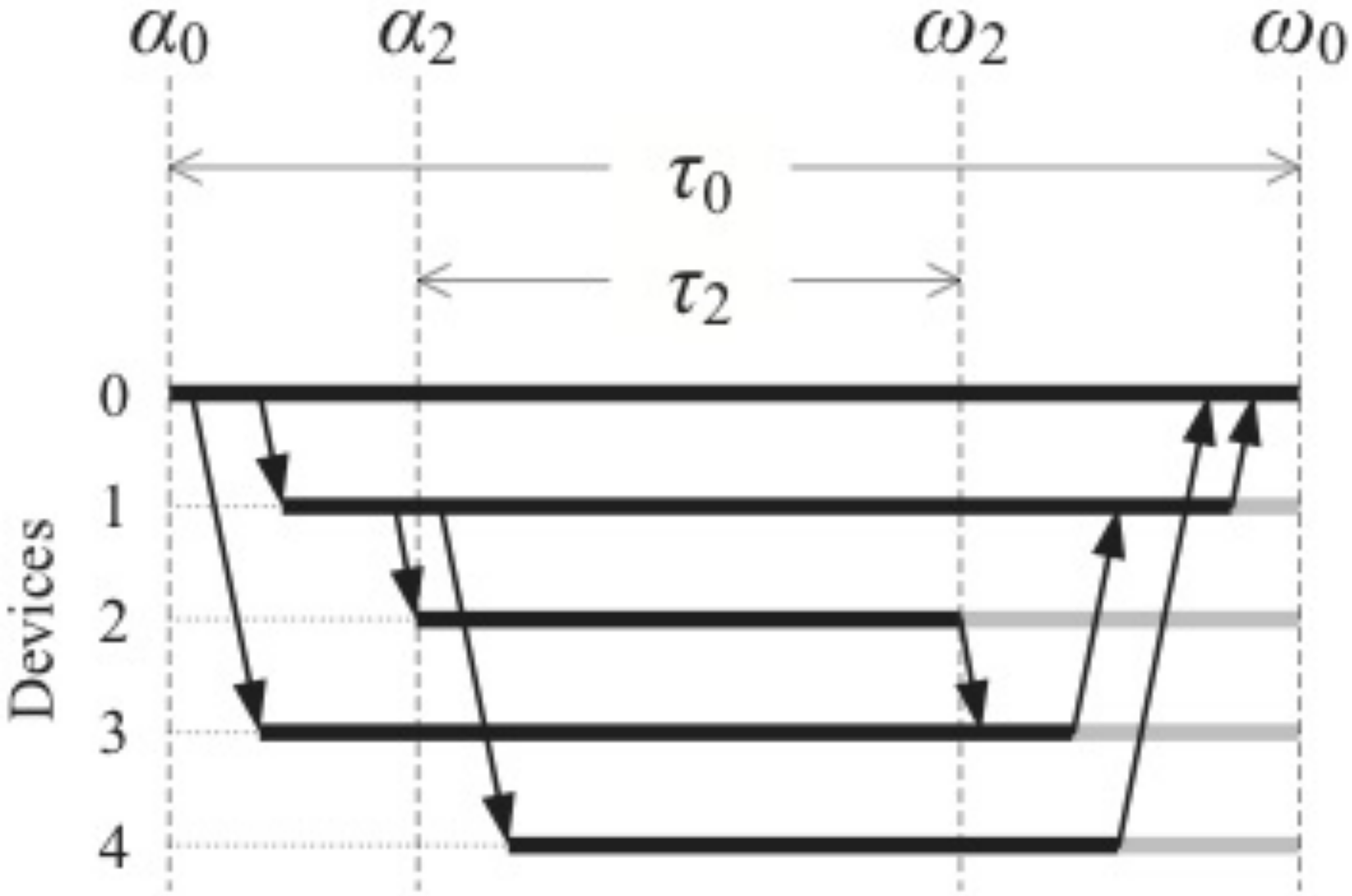
- Current reference implementation:
  - F# targeting .NET platform taking advantage of a vast collection of .NET libraries for implementing D³N primitives
- Future work:
  - Security issues are currently out of the scope of this paper. Executable code migrating from node to node
  - Validate and verify the correctness of the design by implementing a compiler targeting various mobile devices
  - Disclose code in public domain

Part 2 - Task Farming

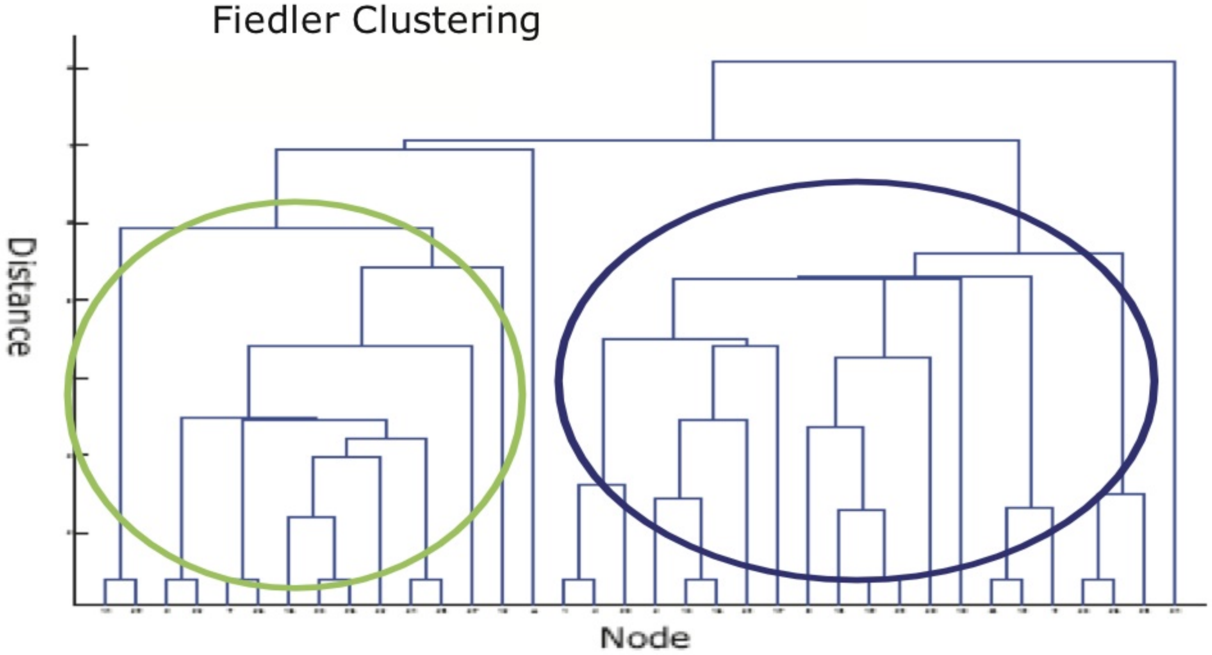


- Sensor readings
- Media
- Web cache
- User knowledge

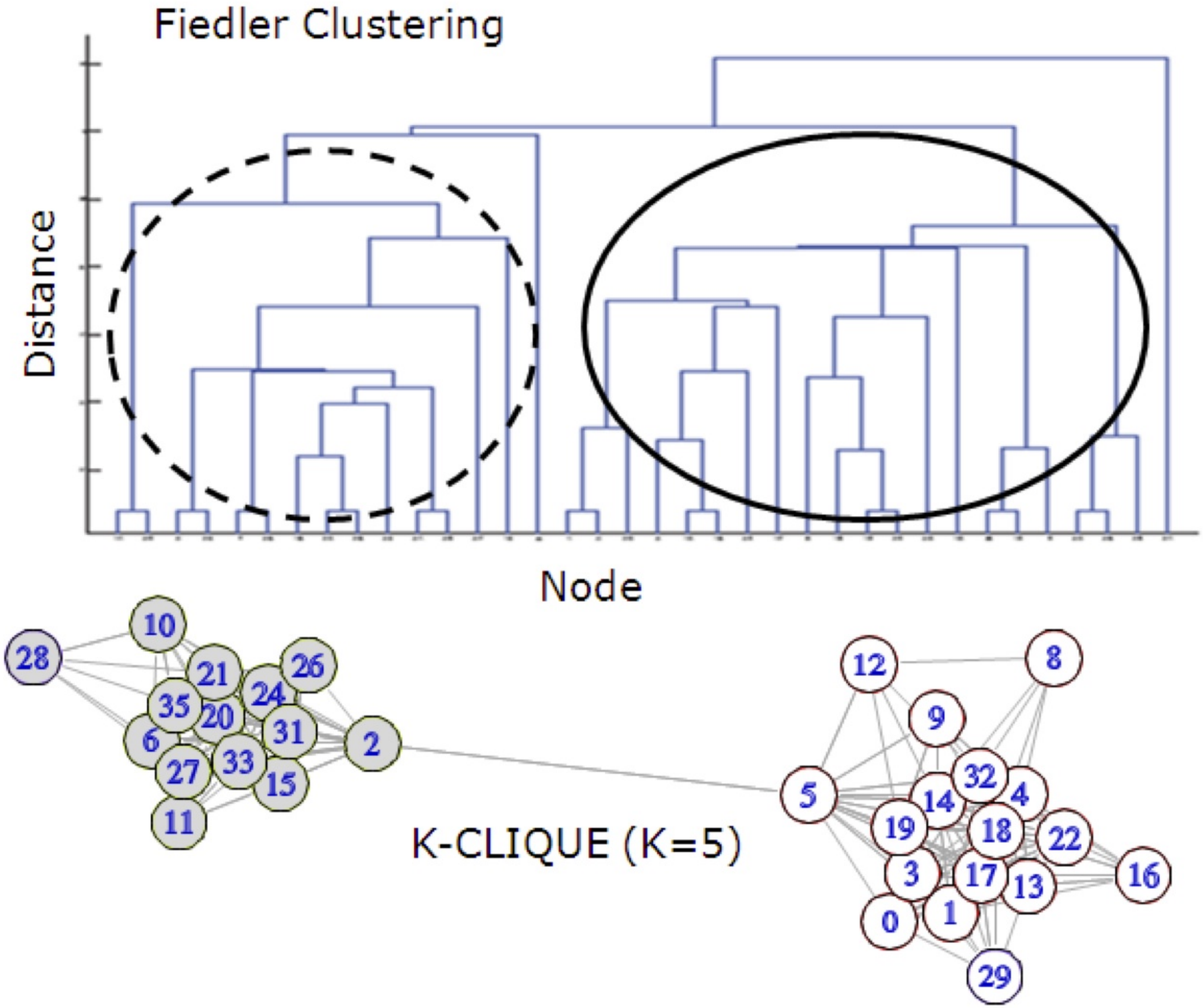
Progress of Computation on Temporal Graph



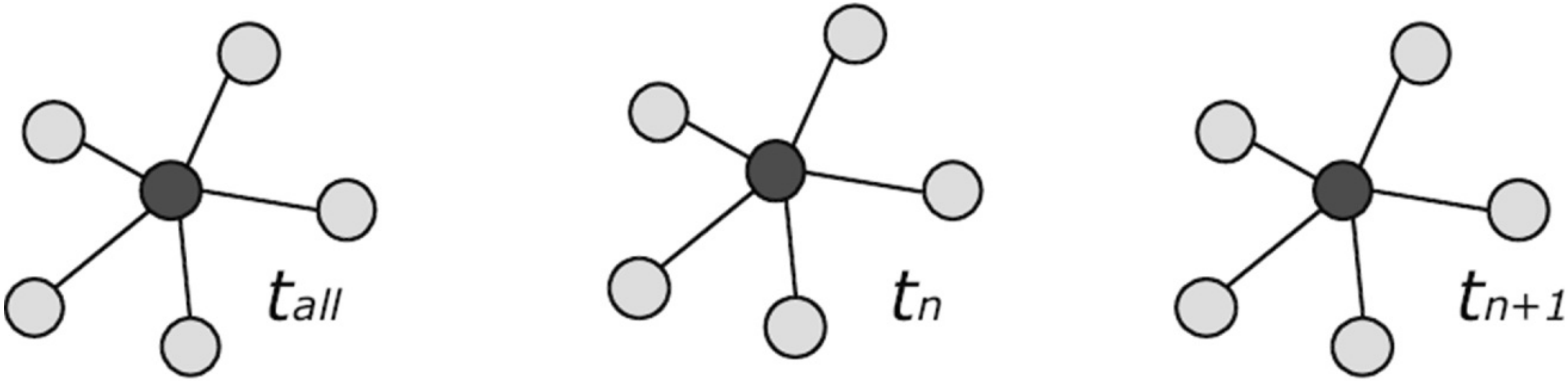
Clustering



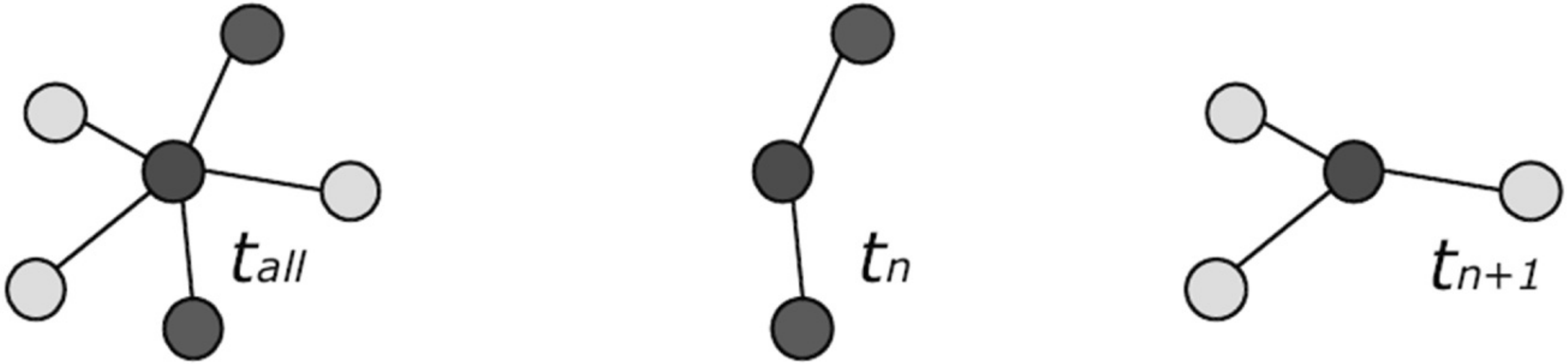
Clustering and Clique Identification



Locality

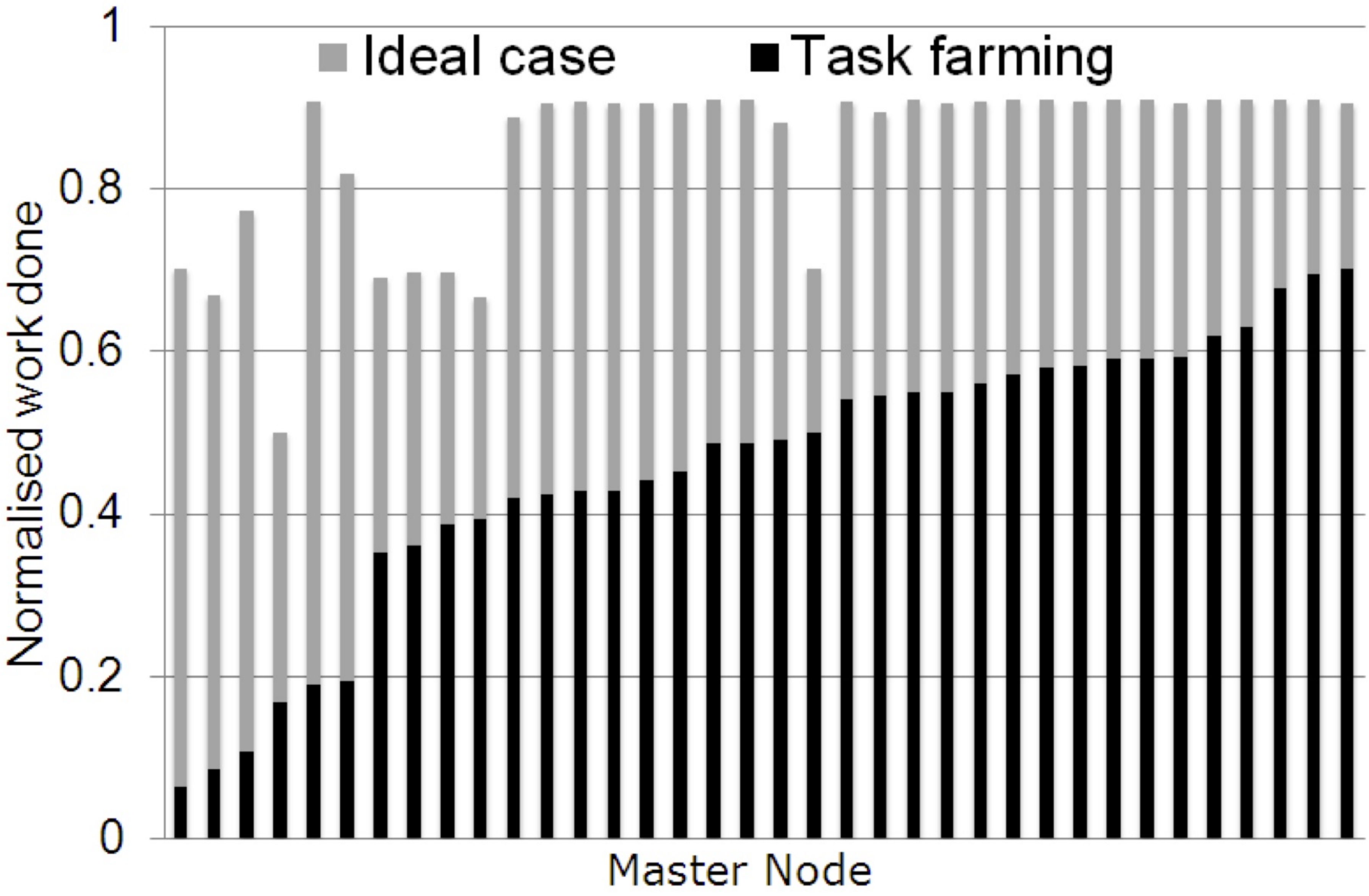


Party Hub: Same Time and Space

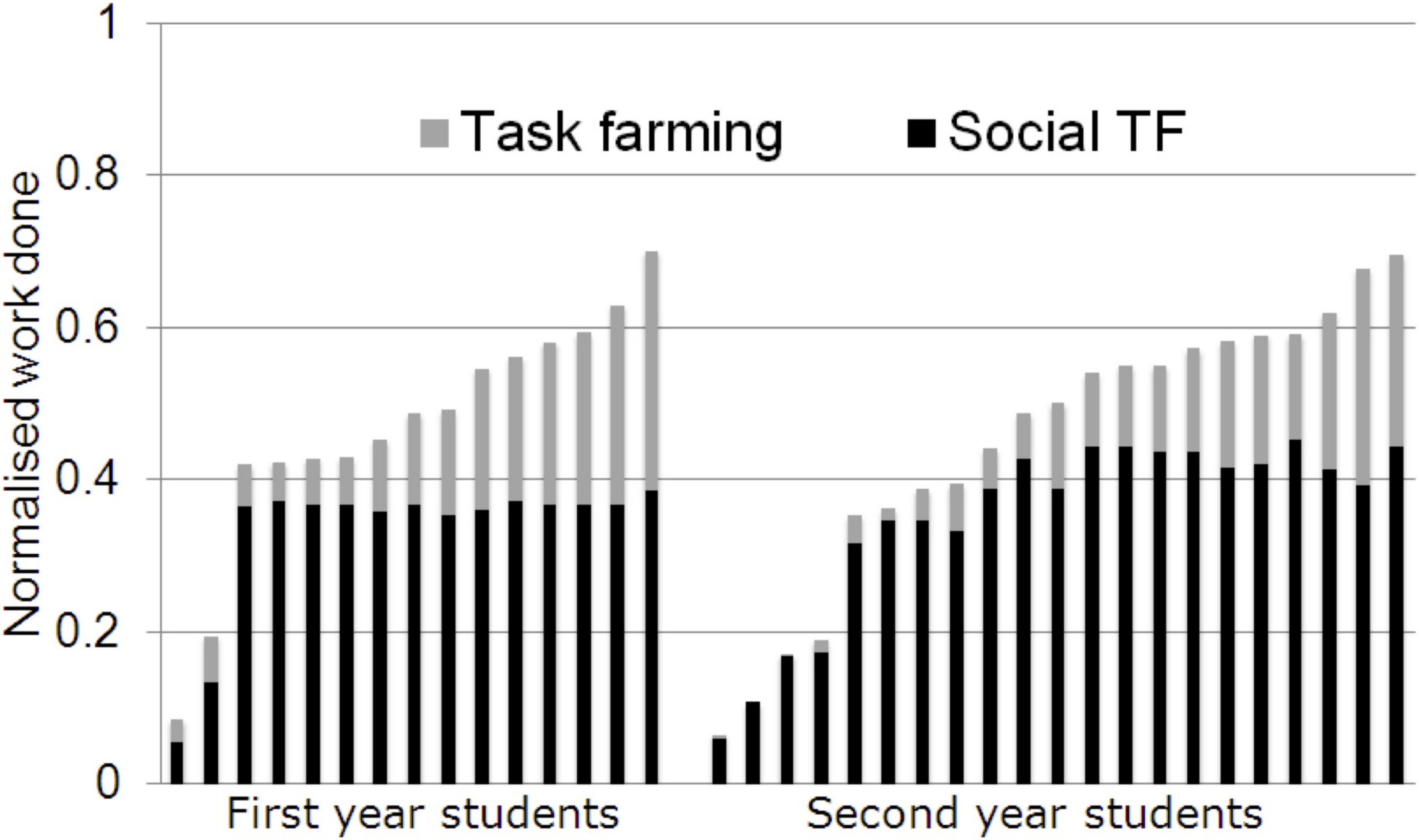


Date Hub: Different Time and /or Space

Work

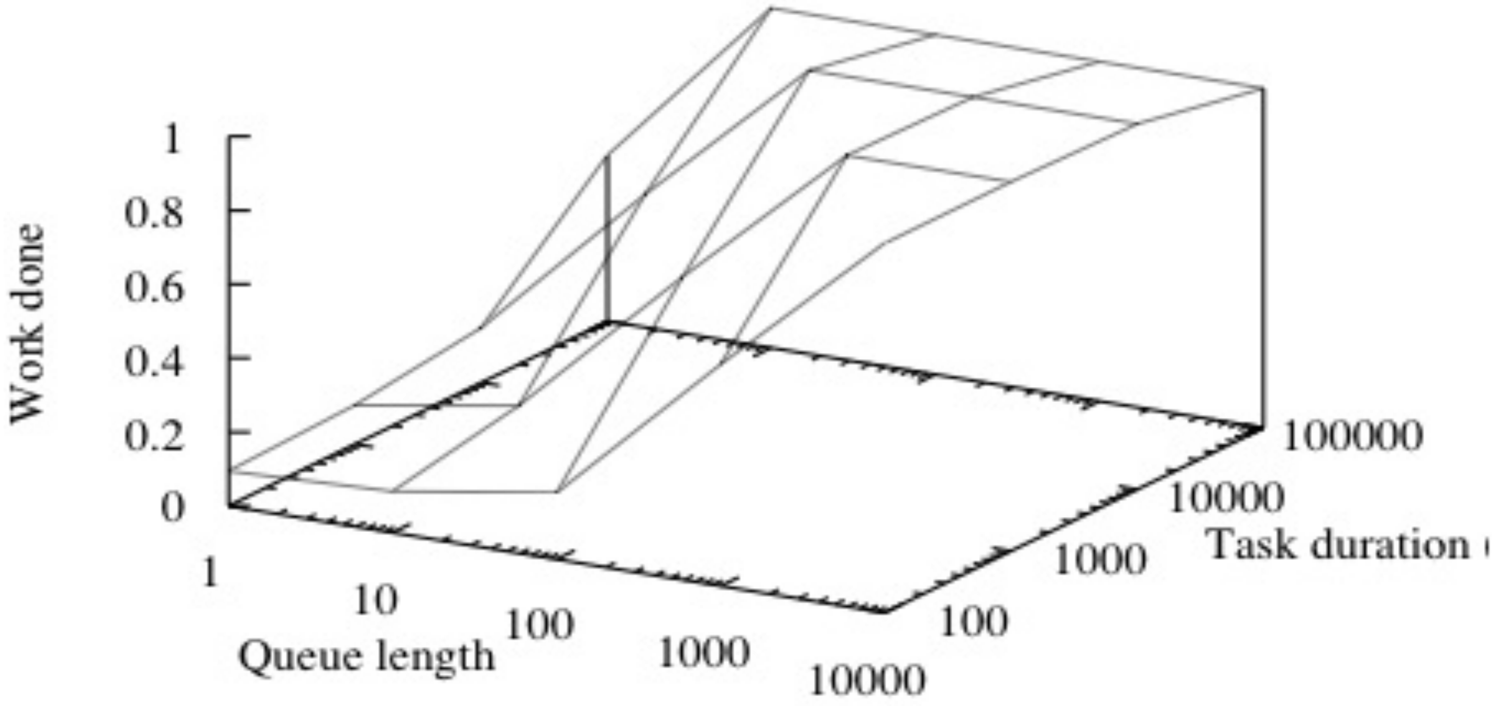

1
0.8
0.6
0.4
0.2
0
Normalised work done
Ideal case
Task farming
Master Node

Social Work

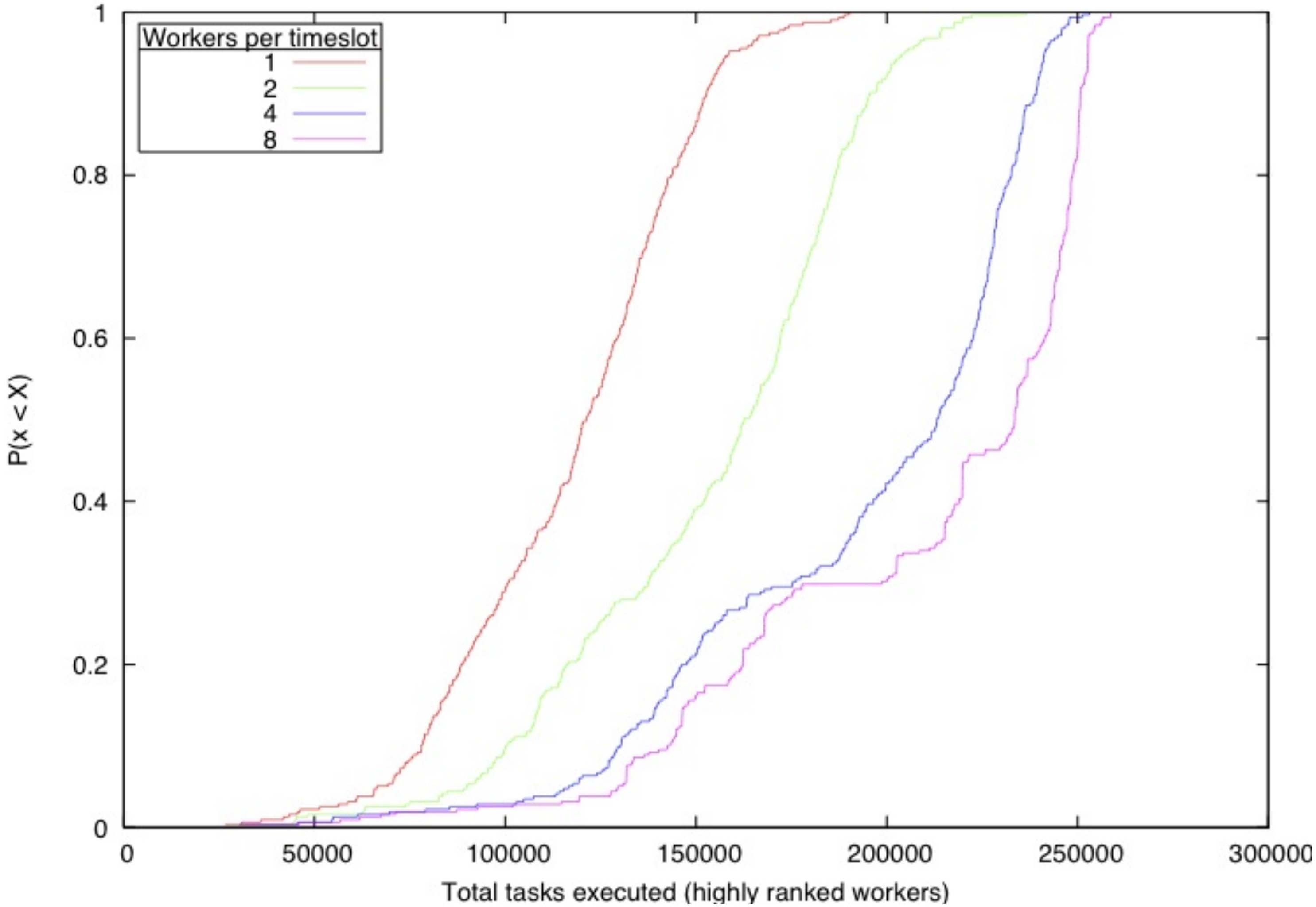


1
0.8
0.6
0.4
0.2
0
Normalised work done
Task farming
Social TF
First year students
Second year students

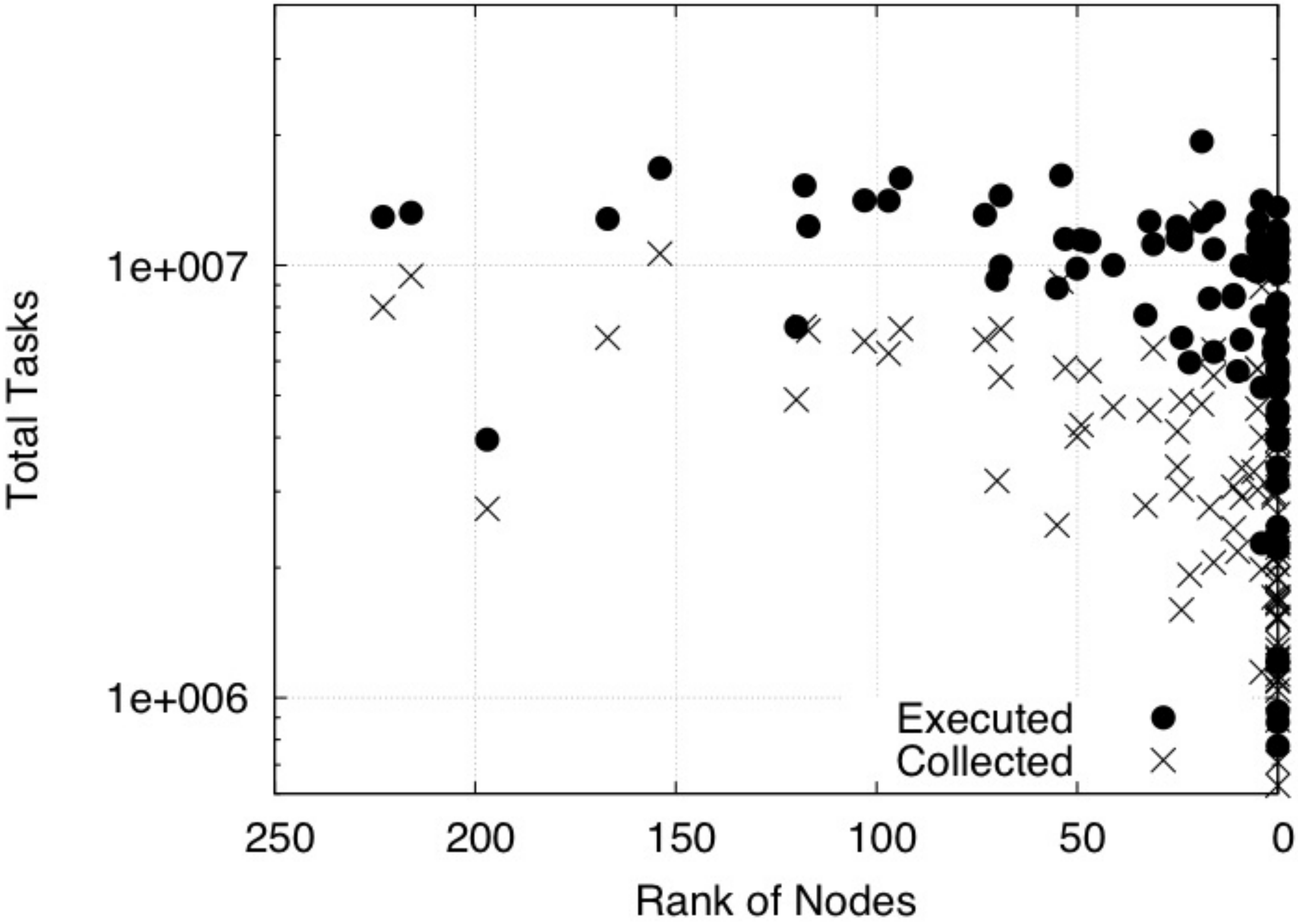
Task Matching



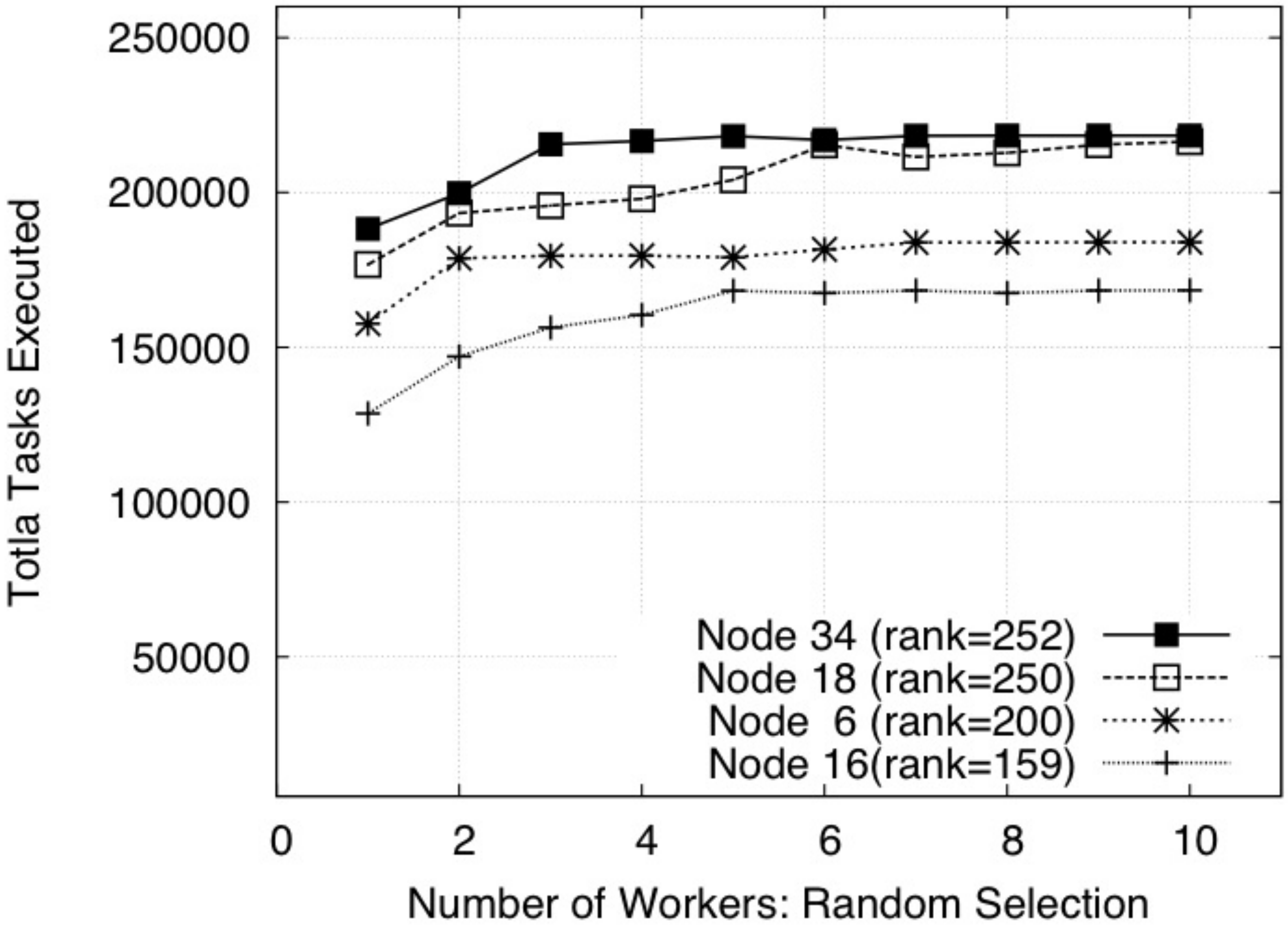
System Level Task Throughput



Rank Effect

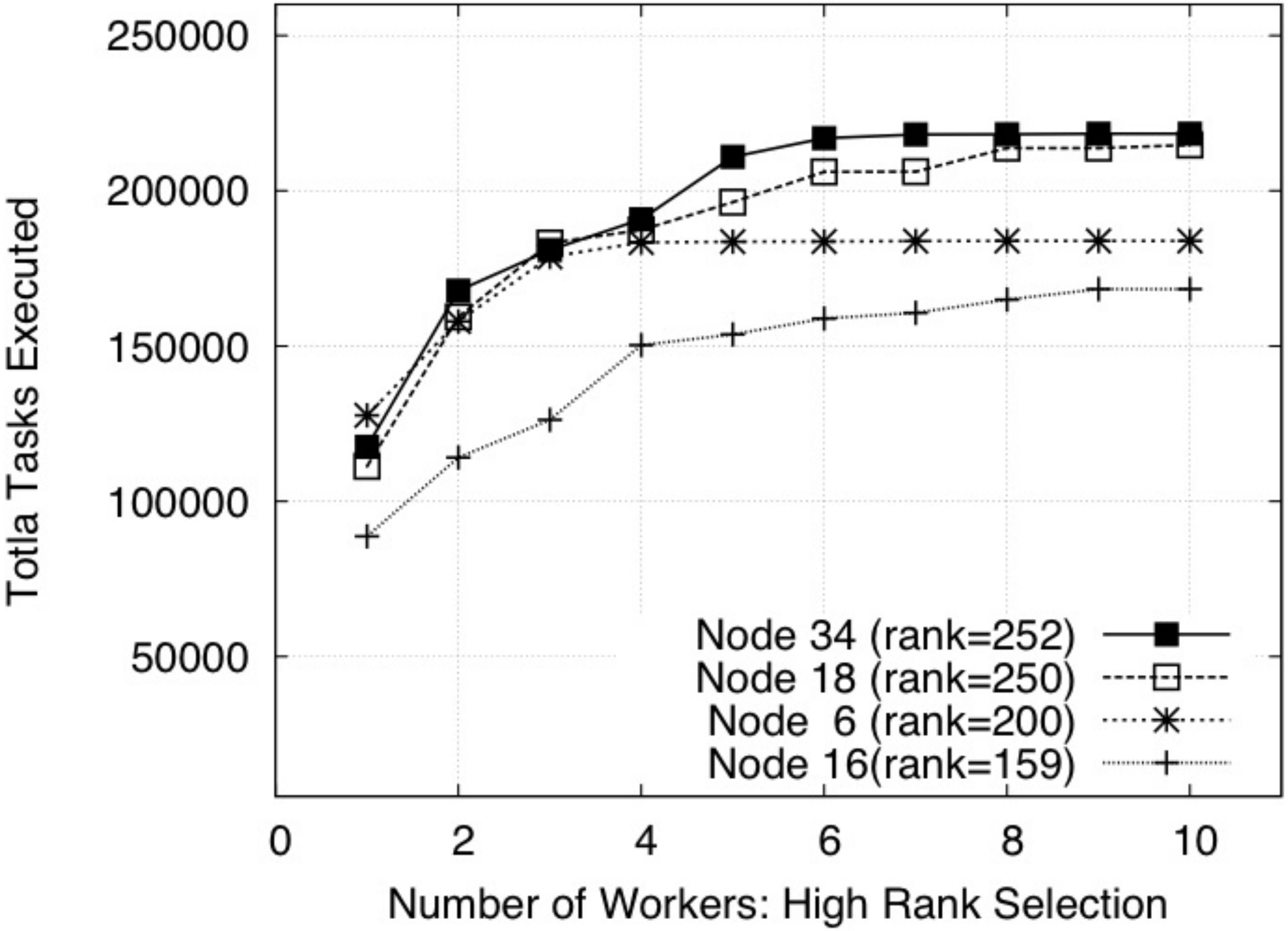

Total Tasks
1e+007
1e+006
250
200
150
100
50
0
Rank of Nodes
Executed ●
Collected ×

Snapshot



Totla Tasks Executed
250000
200000
150000
100000
50000
0
2
4
6
8
10
Number of Workers: Random Selection
Node 34 (rank=252)
Node 18 (rank=250)
Node 6 (rank=200)
Node 16(rank=159)

More Rank Impact



Take Homes

- System Architecture is Data Centric
- Task Farming Can be Done
- No idea if battery use will be too strong disincentive
- Might work if we had data centers in cars :-)
- (Electric cars with data centers could use microgenerators & Batteries to time shift energy as well as data/computation)
- Thought experiment maybe could give insights into normal Cloud system design too - I don’t know though:)

The End

- With much thanks&acknowledgements to
- James Scott, Ebon Upton, Menghow Lim, Pan Hui
- Ioannis Baltopoulos, Shu-yan Chan
- Jing Su, Ashvin Goyal, Eyal de Lara
- Christophe Diot, Augustin Chaintreau, Richard Gass