peer-to-peer
and agent-based computing
P2P Algorithms

UNIVERSITY OF ABERDEEN

A word on algorithms

- An algorithm is a description of a procedure:
  - Not an implementation (but close to it)
  - Not in any particular programming language (but close to some/many of them)
- An algorithm must
  - Have a finite number of steps
  - Each step must be computable within a finite amount of time
  - Have all the information it needs to work
  - Describe all circumstances in which it may run
- Algorithms may need to run forever
  - Loops are not always bad (e.g., your OS)
  - If some algorithms stop then problems arise...

A word on algorithms (2)

- Example: algorithm to multiply two matrices

1. **Read** matrix *A*
2. **Read** matrix *B*
3. $C := A \times B$
4. **return** *C*

A word on algorithms (3)

- Example: algorithm to multiply two matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)
Output: Matrix $C_{p \times r}$ resulting from the product $A \times B$

MatrixMultiply($A_{p \times q}$, $B_{q \times r}$)

```
for i ← 1 to p
    for j ← 1 to r
        C[i, j] ← 0
        for k ← 1 to q
            C[i, j] ← C[i, j] + A[i, k] × B[k, j]
return C
```


A word on algorithms (4)

- How much detail is needed?
  - Depends on who will read the algorithm!
  - Reading a matrix is not a big problem...
- However, if you were asked to propose an algorithm to read a matrix, you would need to
  - Propose a representation for the matrix (line by line, or row by row, XML, etc.)
  - Use known constructs (if-then-else, while loops, etc.) to define the "logic" of the algorithm
- Ultimately, understanding an algorithm is a "human" process and hence subjective
  - What you understand easily may not be so easy for others to understand
  - More details may be required, depending on who will "consume" the algorithm

A word on P2P algorithms

- P2P algorithms are distributed
  - Some of their information comes from other (similar) algorithms
  - No shared memory, though!
  - The only means to obtain information is message-passing
- P2P algorithms:
  - Follow protocols (types & order of messages)
  - Have some "logic" to decide
    - What to do when/if a message is received
    - When to send messages

Algorithm for the discovery of peers

- Flooded request model:
  - How does a peer join a network?
  - What information does it need?
  - What messages does it send/receive?
  - Does the discovery process ever stop?
    - The peer who initiated may stop, but will this "perturbation" create a ripple effect in the network which will never end (that is, it'll go on forever)?
- Very important:
  - Unreliable communication – people you send messages to may never reply to you
  - Don't count on anyone!

Algorithm for discovery of peers (2)

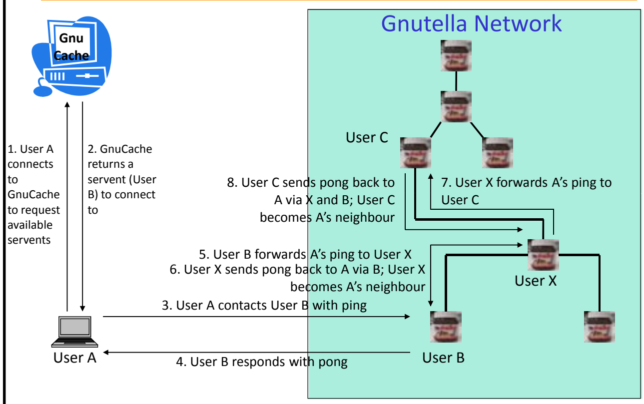
- First attempt

1. begin
2. send ping message to peers
3. receive replies
4. build a list of neighbours
5. end

Which peers? How do we know who is in the network?

What if there are no replies?

Discovery of peers via host caches



Discovery of peers via host caches (2)

- Assumptions:
  - There is a server you first contact to join the P2P network (is this cheating?)
  - All peers will work exactly like you (more on this later...)

Discovery of peers via host caches (3)

1. Introduce yourself to the server
2. Receive N peer ids (name, IP and Port No.)
3. Send ping message to all peers in the list
4. Receive pong messages
5. End (with N neighbours)

Issues:

- What if there are no peers (network is empty)?
- What if no-one replies?
- What if you get a ping message (instead of a pong)?
- What if you get a pong message from someone you did not send a ping message to?
- What if you keep getting pong messages after you completed your desired N neighbours?

Discovery of peers via host caches (5)

Second attempt

1. send message to cache/server
2. receive N peer ids (name, IP and Port No.)
3. PIds $\leftarrow \{P_1, P_2, ..., P_n\}$ // PIds: set of peers
4. for each P in PIds
   1. send ping message to P
5. Ns $\leftarrow$ {} // initialise set of neighbours
6. do
   1. Receive pong from P
   2. Ns $\leftarrow$ Ns $\cup$ {P}
7. until |Ns| = 8 // with 8 neighbours...

Discovery of peers via host caches (5)

Third attempt

1. send message to cache/server
2. receive N peer ids (name, IP and Port No.)
3. PIds ← {P_1, P_2, ..., P_n} // PIds is the set of peer ids/info
4. for each P in PIds
    1. send ping message to P
5. Ns ← {} // initialise set of neighbours
6. do
    1. if a message M (from P) arrives // "non-blocking" receive
    2. case of M
        1. "pong"
            1. if M from someone in PIds then Ns ← Ns ∪ {P}
            2. else ignore M
        2. "ping"
            1. send pong back to P and Ns ← Ns ∪ {P}
        3. otherwise
            1. ignore M
7. until |Ns| = 8 or fedup

Pending issues

- If it finishes with "fedup" and Ns = {}
    - No neighbours...
    - Change the loop to at least 1 peer AND "fedup"
- If it finishes with "fedup" and Ns ≠ {}
    - Peer goes on to search/offer files,
    - Ping message may arrive (forward to Ns)
    - Pong messages may also arrive (add to Ns)

Searching for files

- Parameters
    - Set of neighbours Ns
    - Set of file names Fs2Get
    - Set of files Fs2Share

Searching for files

Share(Ns, Fs2Get, Fs2Share)
1. for all N ∈ Ns do // request to all my neighbours
   1. for all F ∈ Fs2Get do // the files I want to get
      1. send query to N for F
2. while true do // loop forever
   1. if a message M arrives // non-blocking receive message
   2. case M of
      1. "query-ref" // a reply to one of my requests?
         1. if N and F of query-ref one I sent then download F
         2. else ignore M
      2. "query" // someone is asking me for files
         1. if F of query in Fs2Share then send query-ref back
         2. else for each N ∈ Ns do forward query to N
      3. "ping" // someone is pinging me!
         1. if |Ns| < 8 then send pong back and Ns ← Ns ∪ {P}
         2. else for each N ∈ Ns do forward ping to N
      4. "pong" // someone is ponging me!
         1. if |Ns| < 8 and I had pinged this peer, then Ns ← Ns ∪ {P}
         2. else ignore M
3. end

Pending issues

- What about TTL (Time-to-live)?
- What if the algorithm were to be "embedded" in a GUI?
  - New files (to share) coming in
  - New request of files coming in
- What if all your neighbours are disconnected?
  - How could we keep a record on who's connected?
  - How could the information "flow" on the network?

Lessons

- Distributed algorithms share information via message-passing
  - No global variables and no central control
  - Realistic scenarios – no complete information nor full control!
- Algorithms should be, as much as possible
  - Loosely coupled (no "blocking" commands)
  - Information-light (messages carry all info)
- Exceptions in P2P: Peer should ignore the event
- Peers are "stateless"
  - They don't "remember" what they have done
  - All they need to know to decide what to do is encoded in the message