Programação em Sistemas Distribuídos

MEI-MI-MSI

2018/19

3. Models of Distributed Computing

Prof. António Casimiro

Distributed Computing Models

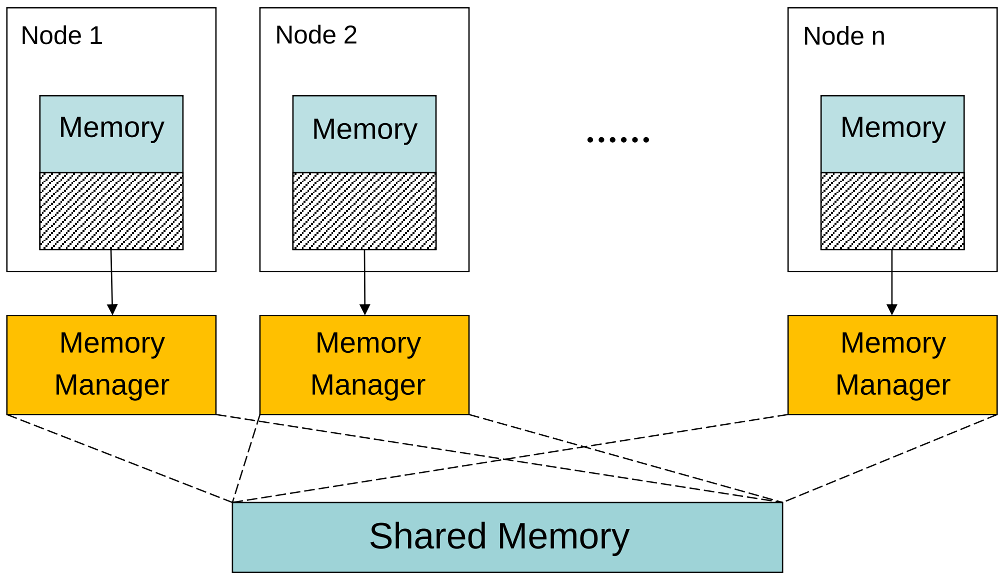
Main models, neither exhaustive nor air-tight

- Client-Server (RPC, RMI, WWW) (Cliente-Servidor)
- Distributed Objects (Objectos Distribuídos)
- **Distributed Shared Memory (DSM, Tuples) (Memória Partilhada Distribuída)**
- Distributed Atomic Transactions (Transacções Atómicas Distribuídas)
- Message-oriented (Message Queue, Publish/Subscribe) (Orientado para mensagens, Fila de Mensagens, Editor/Assinante)
- Stream (Corrente)
- Group-Oriented (Orientado para Grupos)
- Peer-to-peer (Inter-pares)

Distributed Shared Memory (DSM)

Virtues of DSM

- Allowing the shared memory model to be used in systems that don't have physical shared memory
- Allowing all nodes to share a virtual address space (through the shared memory model)
- Allowing communication costs to be reduced by maintaining data copies on the location of access

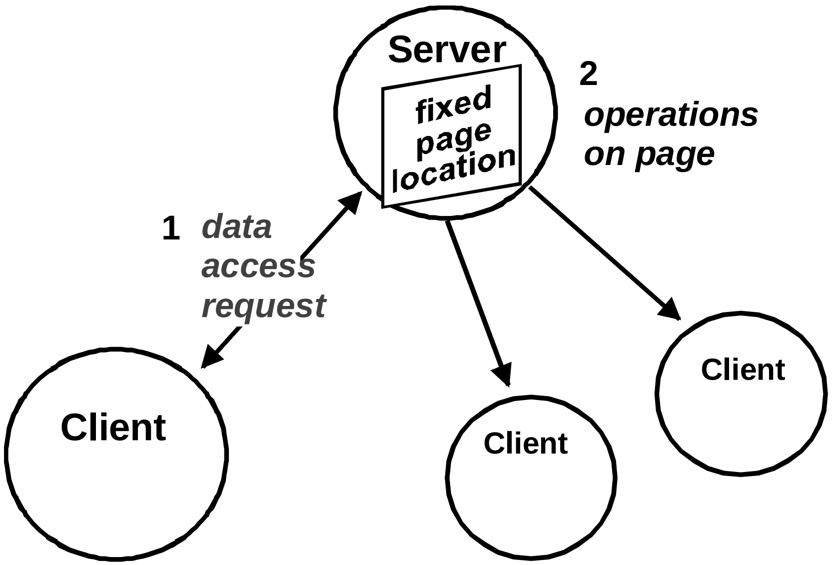


Desirable characteristics of DSM

- Transparency
  - Memory distribution should be transparent to the application and give the illusion of a centralized shared memory system (semantics and performance)
- Traffic optimization
  - Algorithms should minimize number of messages exchanged
- Remote access masking
  - Remote accesses should be hidden from programmer (e.g. node forced to wait for a message from another node in order to proceed)
- False sharing prevention
  - Minimize probability of contention of nodes caused by page granularity (e.g., when two nodes update unrelated variables that, by coincidence, are located in the same page)

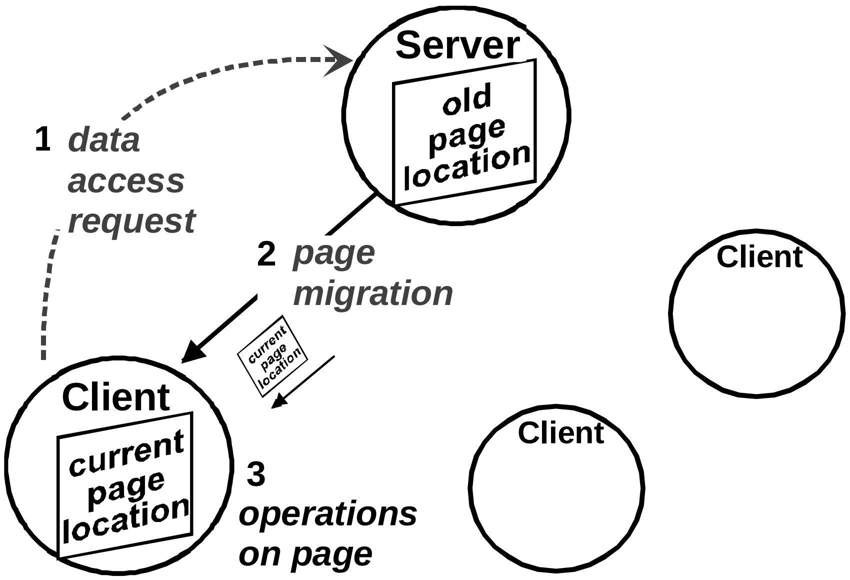
DSM Architectures

- **Centralized server**
  - Naive architecture, for starters
  - Central server holds the memory pages and all data accesses are performed invoking the server
  - Server serializes all requests, thus ensuring strong consistency
  - System behaves just like if a single central memory was available
  - Correctness relies on having a single central memory on the server!
  - **Cons**: since memory accesses are remote, performance is deplorable



DSM Architectures

- **Page migration**
  - One way to improve on central server architecture is exploring locality
  - Instead of forcing all accesses (reads, writes) to execute remotely, migrate a chunk of memory (page, block) from the server to the client
  - Only one node can access the shared data chunk at a time
  - Subsequent accesses to that same page performed locally by the client
  - If another client later wants to access the same page, the page is migrated again
  - Correctness depends of a given page being at a single location at any given point in time
  - **Cons**: high client contention for a page causes it to go back and forth (trashing), performance is affected

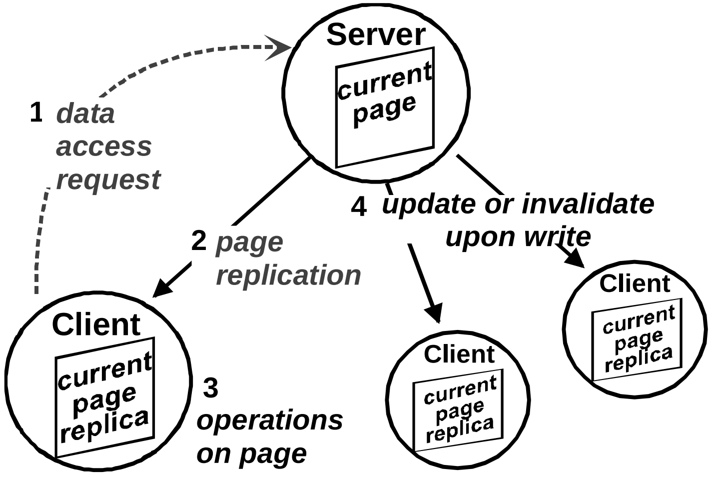


DSM Architectures

- **Read replication**
  - Assume reads are much more common than writes
  - **Read-only replicas** held in several clients
  - **Read-write replica** held in a single clients
  - After a write, all copies are invalidated or updated

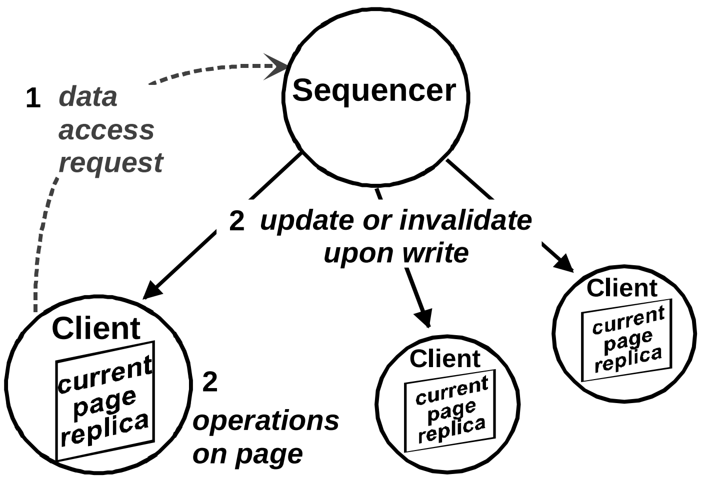
  - **Write-invalidate**: all shared copies are invalidated before write executes

  - **Write-update**: all shared copied are updated with new value



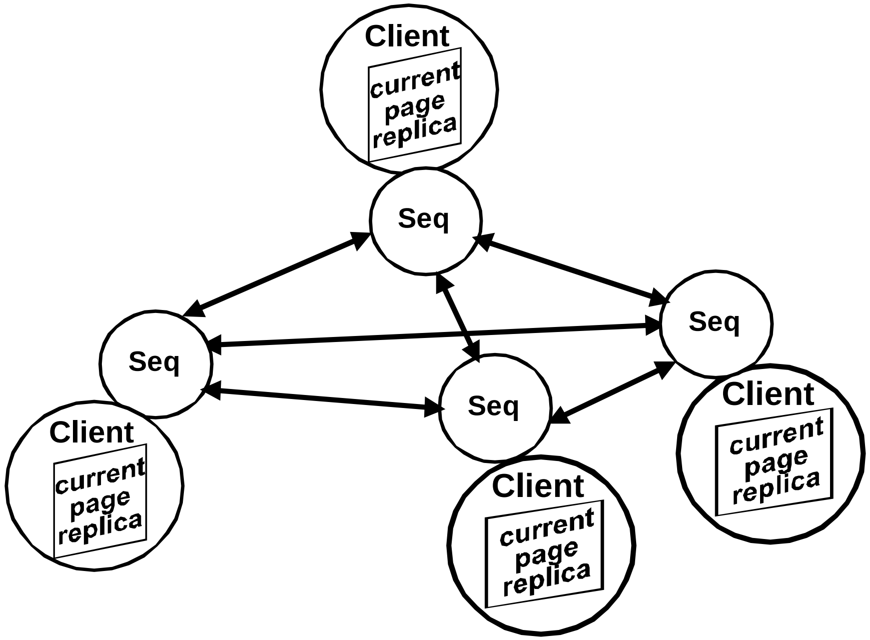
DSM Architectures

- **Full replication**
  - Assume reads are much more common than writes
  - **Read-write permanent replicas** held in several clients
  - Write requests are coordinated by a sequencer: clients perform competitive access requests which are ordered



DSM Architectures

- **Full replication with distributed sequencer**
  - Sequencer may be decentralized, implemented by a distributed algorithm
  - Most elegant and widely used approach



DSM consistency
Fundamental models

- **Strict consistency or atomic consistency**:
  - P1- Every read to a data x returns the value most recently written to x
  - P2- Writes and reads are ordered according to the **real time** order
  - In practice not fully achievable, due to clock uncertainties
- **Sequential consistency**:
  - P1- Every read to a data x returns the value most recently written to x
  - P2- Writes and reads are ordered according to the **program** order
  - Weaker than strict consistency
  - Achievable with total ordering of writes
- **Causal consistency**:
  - Weaker than sequential consistency
  - Only write operations that are causally related must be ordered
  - Therefore, no total ordering is needed for concurrent writes

DSM consistency
Fundamental models

- **FIFO consistency**:
  - Weaker that causal consistency
  - Operations must only be ordered respecting process order
  - Example (assume x=0 in the begining):
    - P1 writes x=1
    - P2 writes x=2
    - P3 can read, in sequence, x=1 and then x=2
    - P4 can read, in sequence, x=2 and then x=1
    - Would this also be a valid sequence if the DSM offers causal consistency?
    - What about if P2 reads x=1 and then writes x=2?
- **Release consistency**:
  - Explicit synchronization with acquire (i.e. lock) and release (i.e. unlock) operations
  - Programs loose transparency, but no DSM mechanisms need to be applied while applications are within critical sections (memory may become inconsistent in these periods)
  - Consistency enforced upon release operations

Tuple Spaces

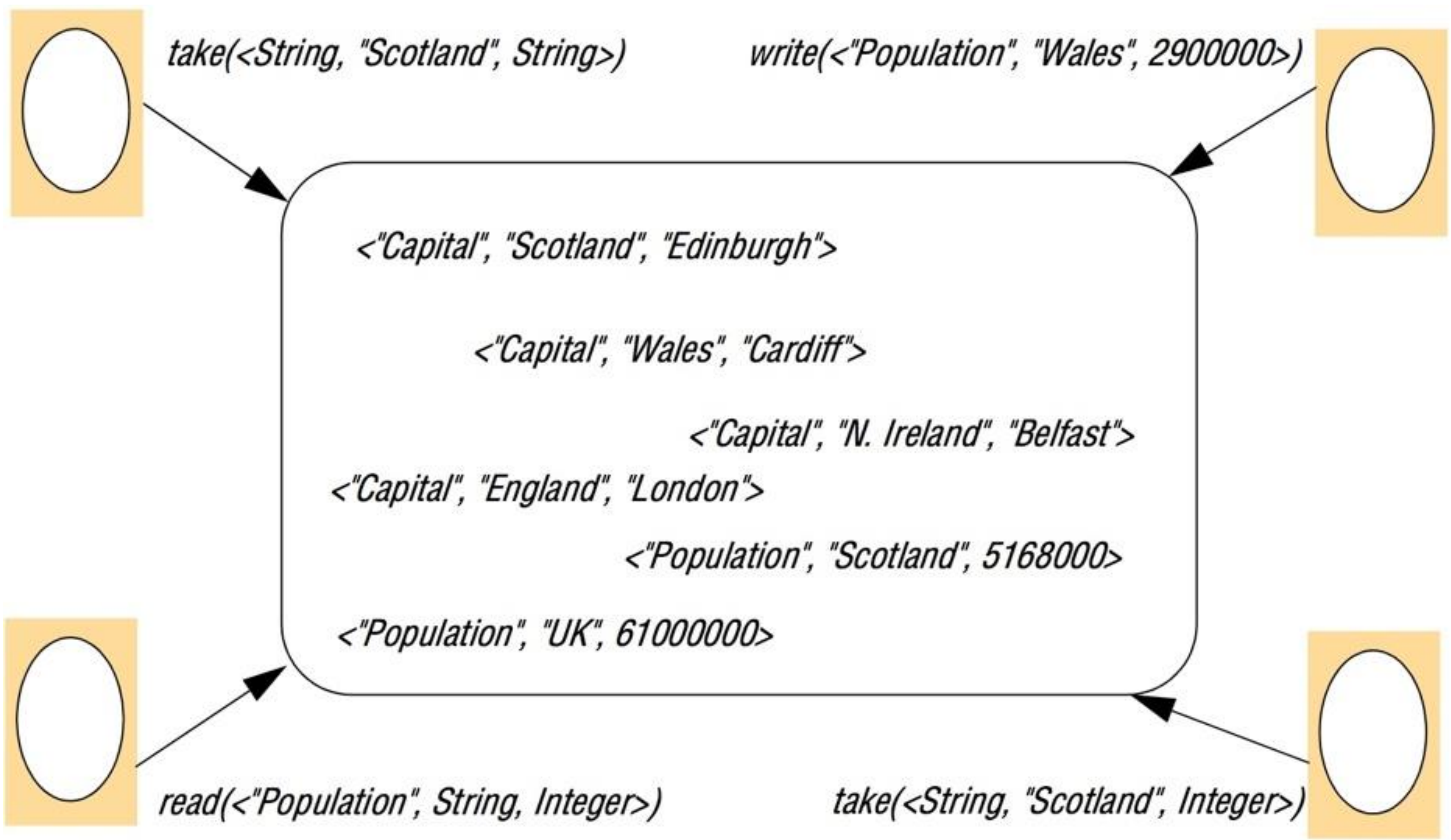
Characteristics of Tuple Spaces

- Processes communicate indirectly by placing tuples in a tuple space, which other processes can read or remove
- Tuples do not have an address but are accessed by pattern matching on content (content-addressable memory)
- Industrial implementations: Sun JavaSpaces, IBM Tspaces

Tuple space
Generic Interface

- ***write(t)***
  – Requesting process inserts the tuple ***t*** into the tuple space
- ***read(tspec)***
  – Requesting process issues read request to tuple space
  – Tuple(s) matching requested tuple specification ***tspec*** are returned, space is not affected
- ***take(tspec)***
  – Requesting process issues take request to tuple space
  – Tuple(s) matching requested tuple specification ***tspec*** are returned and extracted from space

Tuple space
Programming model



Tuple space
Programming model

- Tuple specification
  - Read or take provide a tuple specification tspec and the tuple space returns any tuple that matches that specification
- Blocking
  - Read and take operations both block until there is a matching tuple in the tuple space
- Tuples are immutable
  - No direct access to tuples in tuple space is allowed
  - To modify tuples, processes replace them: take, modify, then write
- Programming examples:
  - take(<String, "Scotland", String>) will match <"Capital", "Scotland", "Edinburgh">
  - take(<String, "Scotland", Integer>) will match <"Population", "Scotland", 5168000>
  - write(<"Population", "Wales, 2900000>) inserts a new tuple in the tuple space with information on the population of Wales