Linda, FT-Linda, and Jini

Prof. Dave Bakken

CptS/EE 562
February 26, 2002
Outline of Lecture & Further Resources

• Linda™
  – http://www.cs.yale.edu/Linda/linda.html

• FT-Linda
  – http://www.cs.arizona.edu/ftol/languages/

• Jini™
  – http://www.sun.com/jini/
  – Core Jini by W. Keith Edwards, Prentice-Hall
  – The Jini Specification by Arnold et al., Addison-Wesley
  – Jini in a Nutshell by Scott Oaks & Henry Wong, Addison-Wesley

• JavaSpaces™
Linda

• Linda is a coordination language
  – Provides primitives to augment an existing computational language such as C
  – Developed at Yale in middle 1980s (David Gelernter)
  – Originally intended for easier parallel programming
  – When distributed, is an example of (what is now called) middleware

• Linda’s main abstraction is tuple space, an unordered bag of tuples
  – Tuple: logical name and zero or more typed values

• Tuple space (TS) is an associative, distributed shared memory
  – Associative: address by content, not location
  – Temporal and spatial decoupling of processes aids ease of use
    • Temporal decoupling: processes don’t have to have overlapping lifetimes
    • Spatial decoupling: processes don’t have to know each other’s identities
  – Tuples are immutable: cannot change in TS, only add and remove
Linda Primitives

• **out**: deposit a tuple into TS
  – `out("N", 100, true);`
  – `out("N", i, boolvar);` // same as above if `i == 100, boolvar == true`
  – `out` is *asynchronous* – process only waits until arguments evaluated, etc., not tuple deposited into TS

• **in**: withdraws matching tuple from TS, based on a template (the parameters), blocks if none present
  – `in("N", ?i, ?b);` // will withdraw one from above (and others!), fill in `i` and `b`
  – `in("N", 100, true);` // same as above, but no variables changed

• **rd**: just like **in**, but tuple is not withdrawn

• **inp**: just like **in** but not blocking: returns “success” flag

• **rdp**: just like **rd** but not blocking: returns “success” flag
Linda Example #1: Distributed Variable

- Initialization: \textbf{out}(“count”, value);

- Inspection: \textbf{rd}(“count”, ?value);

- Updating: \textbf{in}(“count”, ?oldvalue);
  
  // calculate newvalue, maybe \( f(oldvalue) \)
  
  \textbf{out}(“count”, newvalue);
Linda Example #2: Bag-of-Tasks

• Task to be solved is divided into subtasks
• Subtasks placed into TS “bag”
• Pool of identical workers repeatedly:
  – Withdraw subtask tuple
  – Calculate answer
    • May generated new subtasks (“dynamic” if so, “static” otherwise)
  – Deposit result tuple
• Advantages of “Bag-of-Tasks”
  – Transparent scalability
  – Automatic Load Balancing
  – Ease of utilizing idle workstations
• Note: “Bag-of-Tasks” also called “Replicated Worker”
Bag-of-Tasks Worker

process worker
   while true do
      in(“work”, ?subtask_args);
      calc(subtask_args, var result_args);
      for (all new subtasks created by this subtask) // in calc…
         out (“work”, new_subtask_args); // in calc…
         out(“result”, result_args);
   end while
end process

• Problems
  – Lost tuple problem: a failure causes a tuple to be lost
  – Duplicate tuple problem: failure causes subtask tuples to be regenerated
FT-Linda

- PhD dissertation research of Bakken, concluded in 1994
- System model
  - Distributed system with no physically shared memory – only message passing
  - Failure model: fail-silent
    - FT-Linda runtime converts into fail-stop by detecting and depositing a distinguished failure tuple
  - Globally unique logical process IDs (LPIDs)
    - Exactly one for every running process
    - If a process fails, another process may become that LPID
- Main Fault Tolerance Constructs
  - Stable tuple spaces
  - Atomic execution of tuple space operations
    - Atomic guarded statements: all-or-none execution of multiple TS operations
    - TS transfer primitives: atomically move/copy tuples between TSs
Supporting Stable Tuple Spaces

• Support different kinds of tuple spaces
• Tuple space attributes: resilience and scope
• Resilience: **stable** or **volatile**
  – **Stable**: survives $N-1$ failures with $N$ replicas
  – **Volatile**: no survival
• Scope: **Shared** or **private**
  – **Shared**: any process may use
  – **Private**: only the LPID which created it may use it
• TS creation
  – At startup, one **{stable,shared}** TS, $TS_{Main}$, is created
  – $handle = ts\_create(resilience, scope, LPID)$
  – $handle$ is passed as first argument to all FT-Linda TS operations
• “replicated TS”: **shared** resilience
• “local TS” or “scratch TS”: **{volatile,private}**
Atomic Guarded Statement (AGS)

- `<guard → body>`
  - `guard`: `in, inp, rd, rdp, true`
  - `body`: series of: `in, rd, out, move, copy, skip`

- AGS blocks until `guard` succeeds or fails
  - Success: matching tuple found or `true` returned
    - `true` matches immediately
    - `in` and `rd` may match immediately, later, or never
    - `inp` and `rdp` succeed if matching tuple present at start of AGS
      - May be negated with `not` so fails if a match is present
  - Failure: opposite of success, as per above

- Only `guard` may block
  - Exception thrown if operations in `body` block

- TS operations must all be inside an AGS
FT-Linda (Static) Bag-of-Tasks Worker

process worker
  while true do
    < in(TSMain, “work”, ?subtask_args) ➞
    out(TSMain, “in_progress”, my_hostid, subtask_args) >
    calc(subtask_args, var result_args);
    < in(TSMain, “in_progress”, my_hostid, subtask_args) ➞
    out(TSMain, “result”, result_args) >
  end while
end process
FT-Linda (Dynamic) Bag-of-Tasks Worker

process worker

\[ TSScratch = \text{ts}\_\text{create}(\text{volatile, private, my\_lpid}()) \]

while true do

< in(\text{TSMain}, \text{“work”}, \text{?subtask\_args}) \rightarrow

out(\text{TSMain}, \text{“in\_progress”}, \text{my\_hostid, subtask\_args}) >

calc(\text{subtask\_args, var result\_args})

for (all new subtasks created by this subtask) // in calc…

\begin{align*}
\text{out (TSScratch, “work”, new\_subtask\_args)} \\
\text{out(TSScratch, “result”, result\_args) // static: was in AGS}
\end{align*}

< in(\text{TSMain}, \text{“in\_progress”}, \text{my\_hostid, subtask\_args}) \rightarrow

\text{move(TSScratch, TSMain) >}

end while

end process
Monitor Process

process monitor

  while true do
    // one of these failure tuples generated for each replica
    in(TSMain, “failure”, ?host)
    // regenerate all in_progress tuples found from host
    while < inp(TSMain, “in_progress”, host, ?subtask_args)
      out(TSMain, “work”, subtask_args) > do
      noop
    end while
  end process

• Note: monitor process can fail and this still works
Disjunctive AGS

• Disjunctive Form, like a select call:
  
  \[ < \text{guard}_1 \Rightarrow \text{body}_1 \]
  or
  \[ \text{guard}_2 \Rightarrow \text{body}_2 \]
  or
  
  \[ ... \]
  or
  \[ \text{guard}_n \Rightarrow \text{body}_n \]
  >

• Blocks until at least one guard succeeds

• Note: in future slides, we normally omit TSMain for brevity…
FT-Linda Tuple Space Semantics

• Strong \texttt{inp/rdp}:
  – guarantees on \texttt{inp/rdp} matching: first Linda to do this
  – Yale dissertation said it was not possible (even unreplicated!)

• Oldest-matching semantics:
  – Matching tuple which has been in TS longest is returned

• \texttt{out} operations are not completely asynchronous
  – Guaranteed to be found in TS in same order of \texttt{outs} in program
  – Caller of \texttt{out} does not need to block until tuple deposited in TS
    • Just like Linda
FT-Linda Opcodes

• Problem: don’t want to allow arbitrary computation inside a TS operation’s arguments
  – Causes problems for replication if arguments are not the first
  – But we need some computation…
• Solution: allow (binary) opcodes in an AGS
  – PLUS, MINUS, MIN, MAX
• Example: client using actively replicated server
• Server init (once per server replica group):
  – Out(“sequence”, server_id, 0)
• Client calling service
  < in(“sequence”, server_id, ?sequence) ➔
    out(“sequence”, server_id, PLUS(sequence, 1) )
    out(“request”, server_id, sequence, command, args ) >
  < in(“reply”, server_id, sequence, ?reply_args) ➔ skip >
FT-Linda Implementation Overview

• Components
  – Precompiler: translates FT-Linda and C into just C
  – FT-Linda library: implements API for FT-Linda operations
  – TS State Machine: replica of a TS
  – Multicast substrate: deliver AGS operations to all TS replicas in same order (total and atomic)

• Scratch TSs are just a single local copy, others are replicated

• Note: in Linda, associative memory does not cost that much!
  – Patterns (tuple signatures) can be mapped into an integer to hash on
  – Only one variable usually has value specified to match on: hash on it
Jini

• Purpose: allow groups of services and users to federate into a single, dynamic distributed system (Jini community)

• Goals
  – Simplicity of access
  – Ease of administration
  – Support for easy sharing – “spontaneous” interactions
  – Self-healing of Jini communities

• Main operations
  – Discovery: find a lookup service
  – Join: register your service with a lookup service
  – Lookup: find a service in the lookup service
    • Done by type: Java interface type
    • Local object (like CORBA proxy/stub) returned to client
  – Invoke: use the local object to call the service
Other Jini Notes

• Leasing: automatic garbage collection
  – Service granted for a limited period of time: a lease
  – If lease not renewed (it expires), resources freed

• Transactions
  – Two-phase commit
  – Note: Jini, and JavaSpaces are not databases
  – Jini (JavaSpaces) supports full transactions (two-phase commit), “begin transaction” and “end transaction” etc.
  – FT-Linda provides a lightweight (“one-shot”) transaction, not with “begin/end”, but Atomic Guarded Statement with carefully limited actions allowed
    • This is so AGS info can be packed into one multicast message and performed with just that message delivery

• Events
  – Can register for callbacks for events of interest
Jini Example

• Start: one service – lookup – running on network

• Printer starts up
  – Finds lookup service
  – Registers self with lookup service (no user intervention)

• Laptop with word processor enters room
  – Word processor finds lookup service
  – Word processor looks up printer
  – Word processor can also optionally
    • Register to get callback if printer goes away
    • Register to get callback if a new printer registers itself
  – Word processor invokes printer (sends it a printer job)
    • Printer (not word processor) controls dialog box – only it knows what it
      should look like, perhaps in ways not known when word processor made
JavaSpaces

• Jini is built on top of JavaSpaces!
• JavaSpaces is based on Linda!
• Main JavaSpace (JS) operations
  – Add an Entry object into JS
  – Read an Entry object from JS
  – Remove an Entry object from JS
  – Register as a listener of an Entry object
JavaSpace Differences from Linda

• Strong typing
  – Can have multiple JS (Java) types per Linda pattern
• Entries are objects, so they can have methods (behavior)
• Leasing
• Multiple JSs possible
  – Not true for first Linda implementations
JavaSpaces Replicated Worker Example

• (From “JavaSpaces Principles, Patterns, and Practice”)

Public class worker {
    for (;;) {
        Task template = new Task(…);
        Task task = (Task) space.take(template, …);
        Result result = compute(task);
        space.write(result, …)
    }
}

How does a JavaSpaces transaction work with this??
Other Jini Notes

• Jini’s competitor at Microsoft is “Universal Plug and Play”
• Jini-related distinguished speaker was here April 28 2001:
  – Jini-like research prototype system, Aladdin, from Microsoft Research, but where devices do not have to be smart (just configurable)
  – Speaker: Yi-Min Wang
    • Well-known fault-tolerance guy
    • DCOM bigot (Bakken is a CORBA bigot…)