Transparent Distributed Threads for Java

Workshop on Java for Parallel and Distributed Computing

IPDPS 2003
April 22-26, Nice, France

UNIVERSITY OF KARLSRUHE
Computer Science Department

Thomas Moschny
Institute for Program Structures
and Data Organization · Prof. Tichy

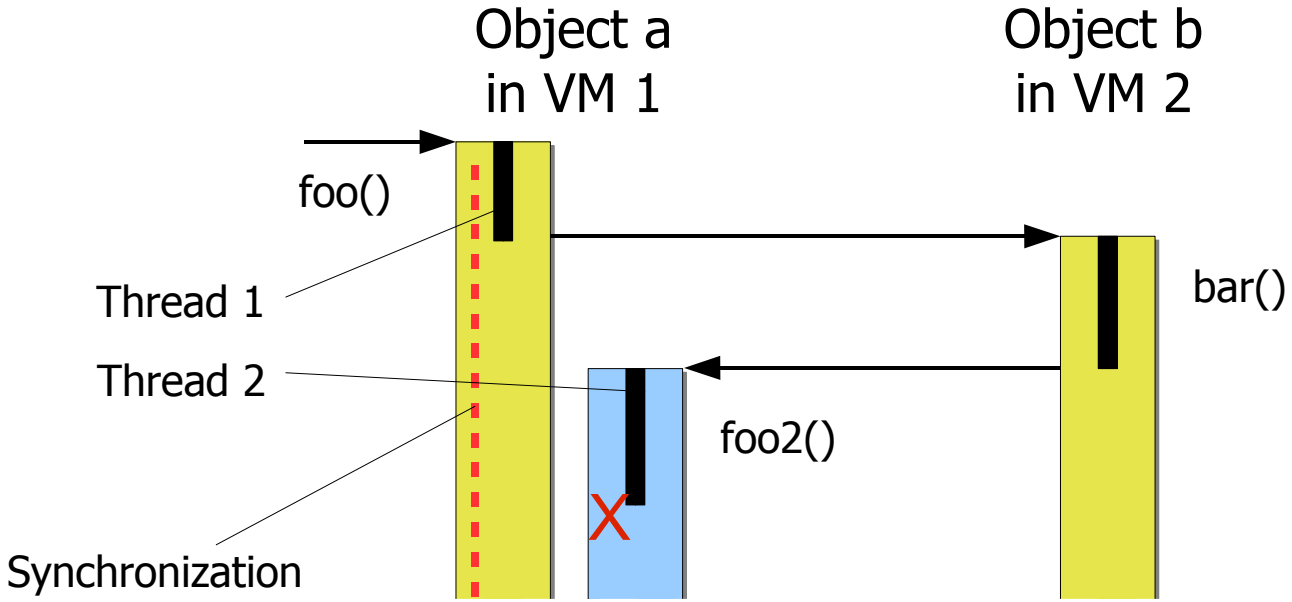
Context

- JavaParty
  - extends Java:
    - Keyword `remote` denotes remote classes.
  - provides a single system view of a cluster with multiple VMs.
  - transforms application code into pure Java with RMI and JavaParty-runtime calls.
- KaRMI
  - efficient RMI designed for clusters
  - implements fast object serialization
  - supports multiple transport technologies e.g. for Myrinet (GM / ParaStation)

Distributed Threads

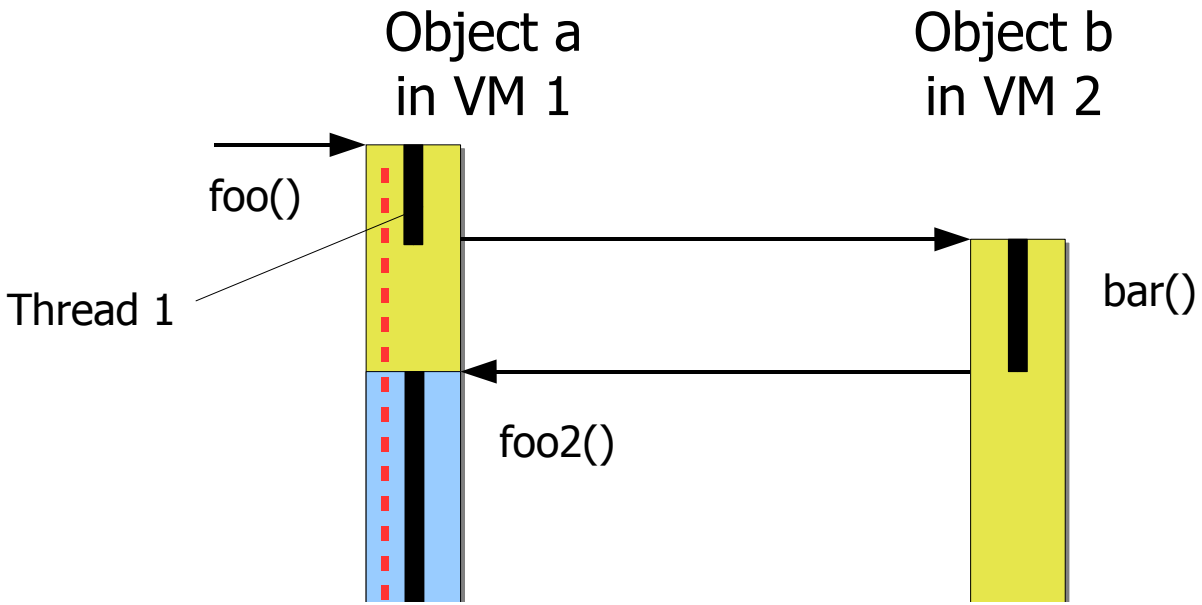
- In a remote method call, the point of execution moves from thread t_1 to another thread t_2 in a different VM.
- t_1 and t_2 can be seen as segments of a the same distributed thread.
- However, distributed threads are not fully transparent:
  - Locks are not reentrant on recursion.
  - Monitors of remote objects cannot be acquired.
  - Signals are not forwarded.
- We present solutions for all three problems.

Synchronization Reentrance (1)



- Consider this situation:
  - `a.foo()` calls `b.bar()` while holding a lock.
  - `b.bar()` calls `a.foo2()`
  - `a.foo2()` tries to obtain the same lock
  - → a deadlock occurs because `foo2()` is executed by thread 2

Synchronization Reentrance (2)



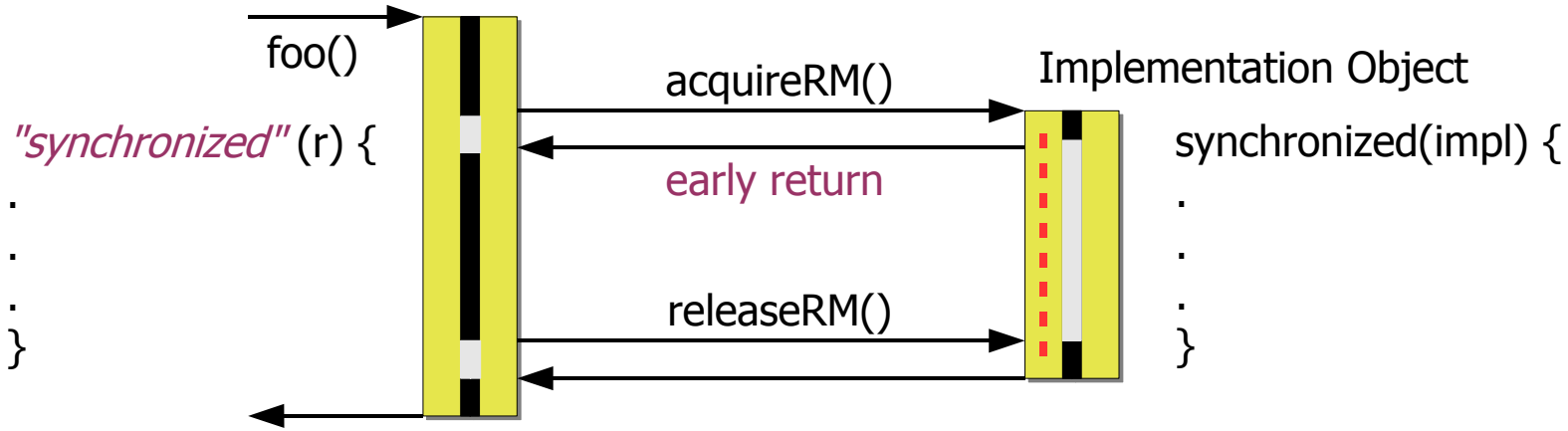
- Re-use of originating thread 1 avoids the deadlock.
- Strategy: every distributed thread has at most one local representative that executes all local segments.
  - A unique thread ID is needed to identify the distributed thread.
- This solution requires changes in RMI.

Synchronization on remote objects (1)

- In Java, blocks can be synchronized on any non-null object.
- This has undesired effects, if the object is actually a handle to a remote object:
  - multiple threads can „lock“ the object simultaneously using different proxies
  - inter-thread communication using `wait()`/`notify()` does not work as expected
- → For regular RMI, synchronized blocks on remote objects are illegal.

Synchronization on remote objects (2)

- KaRMI extension:
  - Facility for acquiring remote monitors:
    `acquireRM()` and `releaseRM()`
  - Instruct the local representative at the remote object's home node to obtain the lock
  - Both methods actually form ***one*** remote call with early return capabilities.



Synchronization on remote objects (3)

- Method does not work well for local objects, because there is no early return in standard Java.
- Standard Java synchronization on the object itself should be used instead.
- This transformation can be performed by the JavaParty compiler.
  - Resulting code depends on KaRMI.

```
Remote obj;
if (isLocatedRemotely(obj)){
  Object rma = aquireRM(obj);
  try {
     // code
  } finally {
    releaseRM(rma);
  }
} else {
  Object lock = getImpl(obj);
  synchronized (lock){
    // code
  }
}
```

Distributed Thread Control

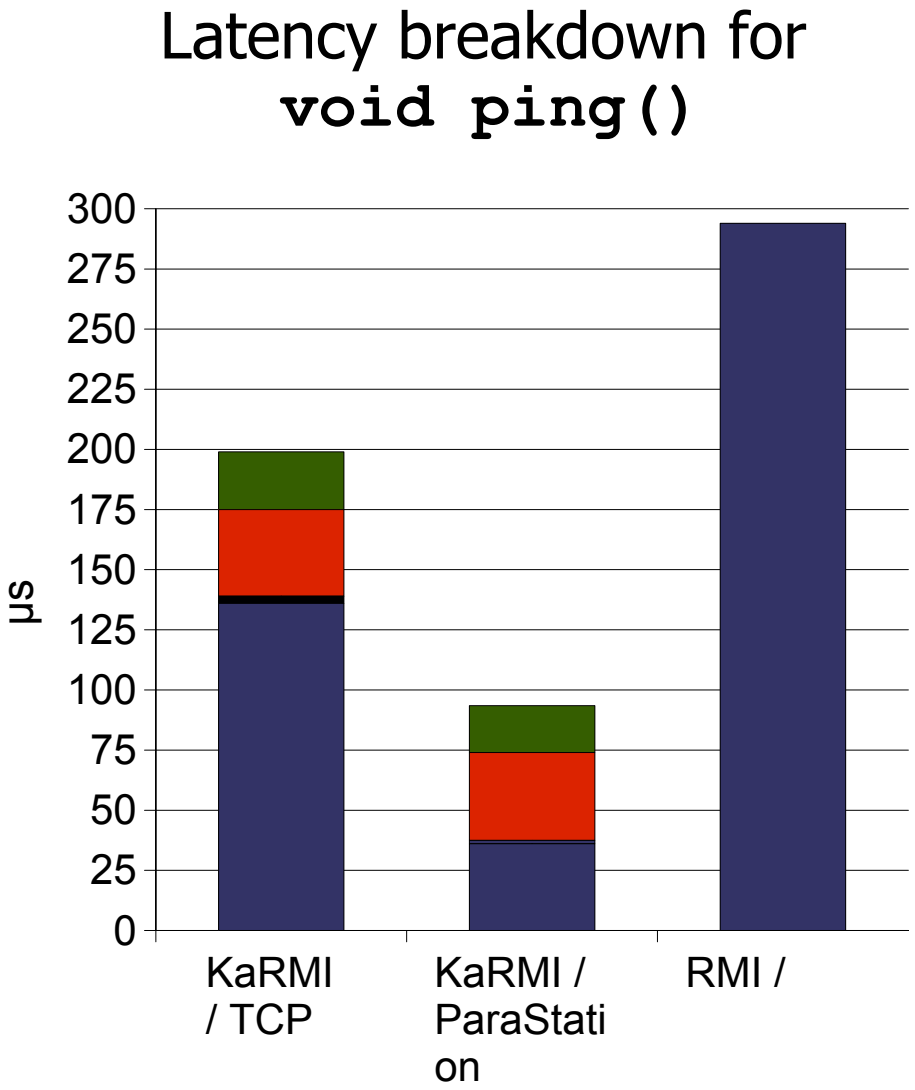
- An interrupt sent to a currently inactive segment of a distributed thread waits locally for the return of the pending communication.
  - The remotely called method may `wait()`, so the interrupt gets never delivered.
- KaRMI extension:
  - Forward an interrupt along remote method calls up to the active segment.
  - The interrupt request is additionally stored locally, because the remote method call may return before the request reaches the foreign node.

Implementation

- During a remote call, a segment of a distributed thread must perform two tasks:
  - wait for the completion of the remote call
  - be attentive for incoming recursive calls of the same distributed thread and for interruption
- In standard `java.io`, communication operations are blocking.
- → In KaRMI, communication is done by a separate thread.
  - This causes additional local inter-thread communication for every remote call.

Evaluation

- remote call
- 1.5-3µs for maintenance of global unique thread identifiers
- ≈36µs on client side for additional inter-thread comm.
- if there is already one, delegation to local representative takes ≈25µs on server side



Conclusion and future work

- Significant enhancements to Java/RMI
  - Transparent distributed threads
  - Synchronization on remote objects
- Minimal overhead
  - only ≈2µs in remote calls for maintenance of global thread id
  - 30% overhead still yields a latency for remote `ping()` that is 40% smaller than that of RMI
- Plans
  - We expect to be able to eliminate client side thread-thread communication by means of java.nio instead of java.io.