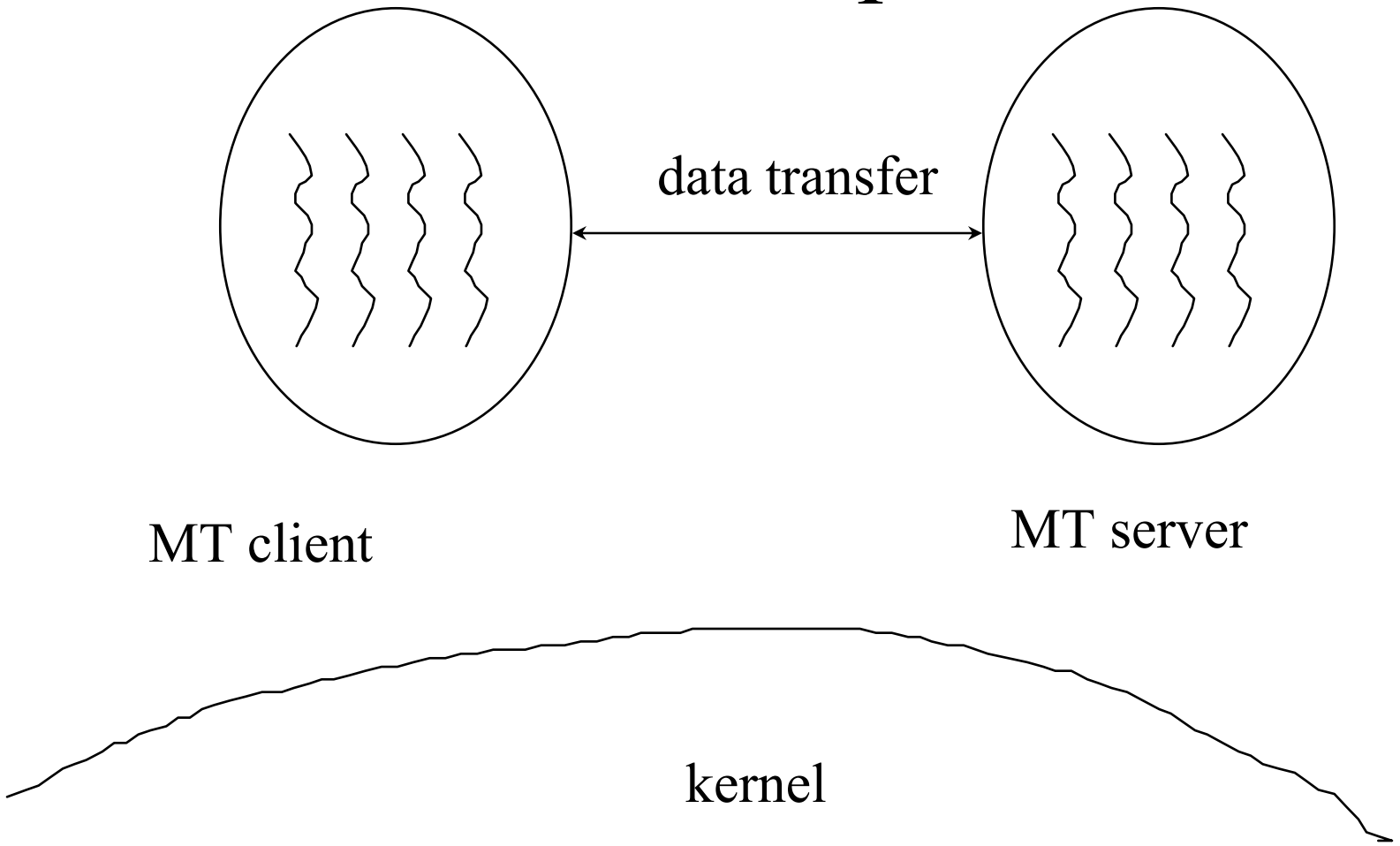
# User-Level IPC for SMPs

- based on paper “User-level Interprocess Communication for Shared-Memory Multiprocessors”, Bershad et al.
  - mixed discussion about user-level threads and user-level RPC

# Cross address space calls



- communicating single/multi-threaded processes on a SMP – multiple address spaces
- small kernels – functionality implemented in modules outside kernel, that need to communicate efficiently
  - common case of communication is not across the net, but across address spaces!
- In general, we want to have separate address spaces
  - promotes modularity, fault isolation, flexibility, extensibility, protection…
  - need to cross address spaces, but that can hurt performance

- ideas in the paper – optimize cross-address space communication – user level thread management with user level communication

# Inter-Process Communication

- needs to be efficient – performance of systems depends upon fast IPC mechanism
- typical ‘local’ IPC mechanisms
  - shared synchronization variables
  - shared memory
- typical ‘local’ or ‘remote’ IPC mechanisms
  - message passing
    - both, including message queues, but also unstructured communication
    - remote procedure calls (RPC)
- our discussion will concentrate on ‘local’ communication

# Remote Procedure Calls

- In RPC, communication between address spaces is structured similarly to procedure calls
- An RPC runtime hides from higher layers address crossing, type checking, procedure parameter and result transfers, etc…)
- From the caller's perspective, RPC are synchronous calls – can have different failure semantics
- details in following lecture

- RPC, and all communication, is closely related to thread management:
  - e.g., a server thread may need to be scheduled when it receives a message; a client thread may need to be de-scheduled while it is waiting for a response
  - we mentioned previously communication and synchronization – two faces of same coin.

# URPC

- standard RPC is kernel-based
  - this is overkill for multi-threaded applications: argument similar as for user-level threads: cost of trapping in the kernel, switching memory management context…
- URPC is specialization of RPC for a SMM
- Characteristics summary:
  - shared memory is used for passing arguments and results, without kernel invocation
  - address-space switching can be avoided or made less frequent by lazy address-switch
  - valuable for SMPs but also for uniprocessors running multithreaded programs

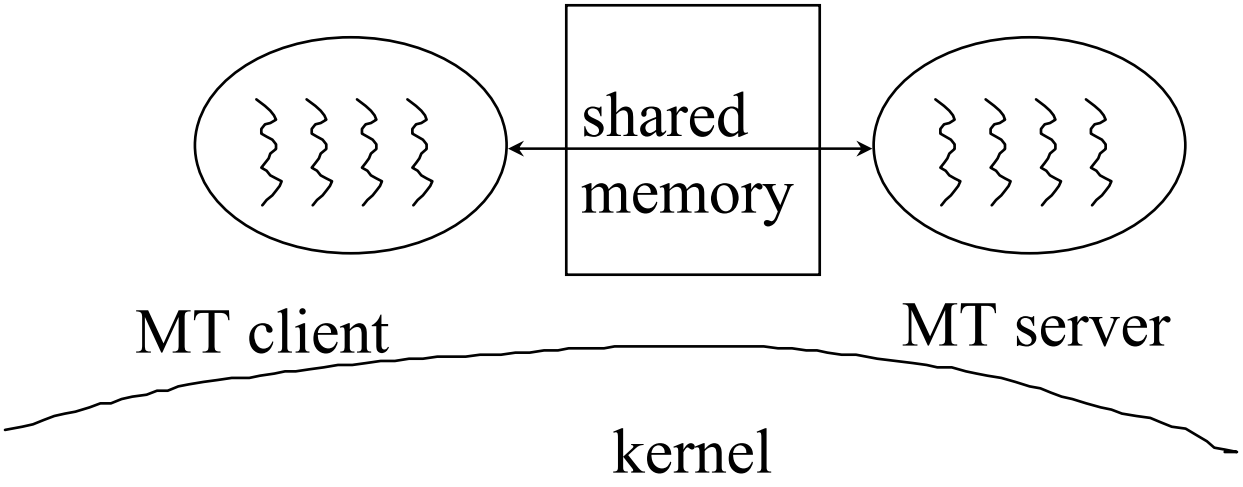
# IPC requirements

- processor reallocation
- thread management
- data transfer

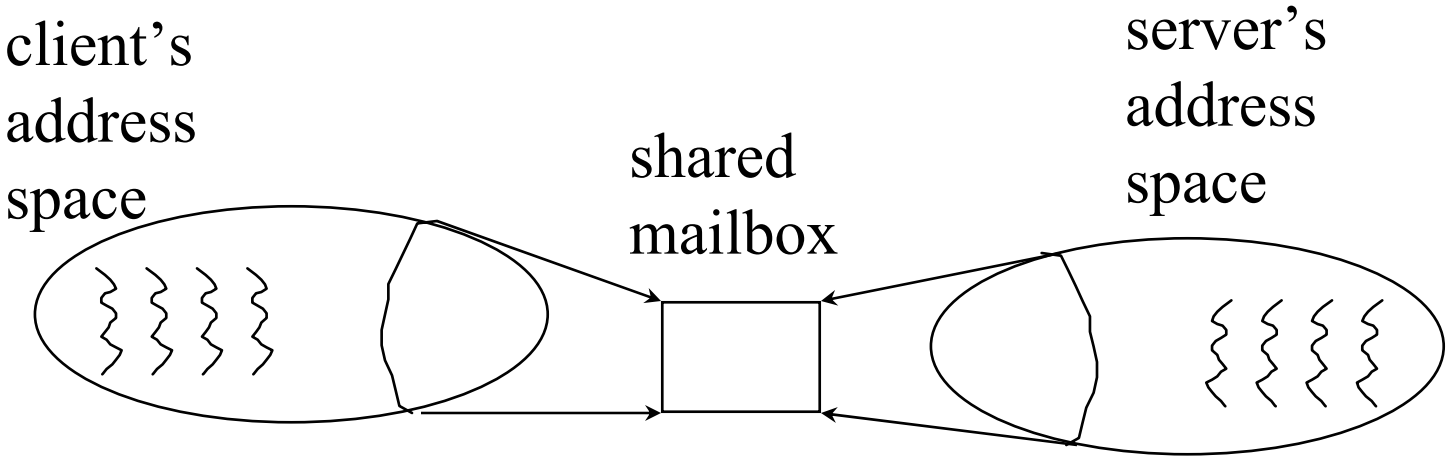
How to do these without kernel involvement?

- first two effect a control transfer from one address space to another
- is kernel involvement needed for all the three requirements?

- only processor reallocation needs kernel involvement
- thread management can be done at user level
- what about data transfer?



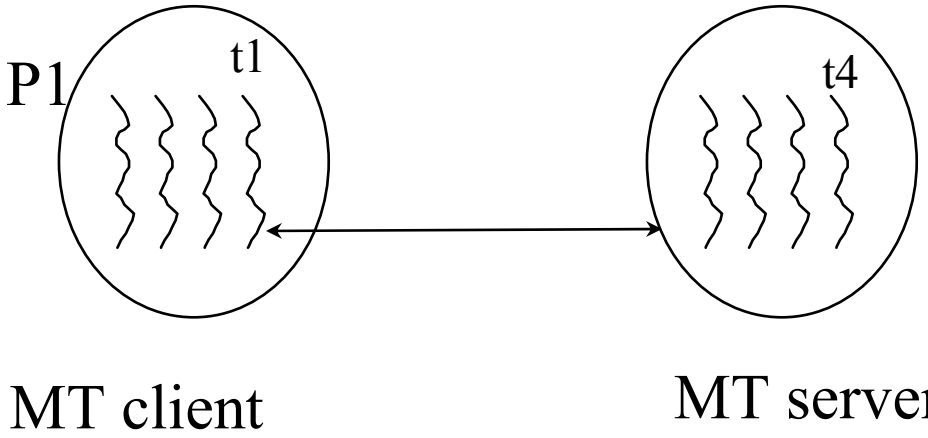
# Integrity of data?




- who should do the mapping?
- who should check what is being passed in the mailbox?

- authentication implied statically
  - mapping done once by kernel pairwise between client and server
- correctness checked dynamically
  - on each call-return between client and server by URPC runtime (runtime responsible for putting data in memory buffers, inspecting type, range, etc…)
- upshot?
  - cross address space calls can be implemented without involvement of the kernel – both send and receive can be done within the user-level library…
  - right?

- not quite…
  - what about processor reallocation? (address switching)




- t1 blocked on call
- why not pick a thread t4 in server to run on P1 to execute the call?
  - kernel op to change VM context
  - cache and TLB performance
  - want to avoid address switching

- solution:
  - give P1 to some other ready in client (thread management at user level)
  - if server is already executing on P2 then use it to run the server thread to handle the call

  i.e. keep the OS out!
- always possible to do this?
  - if an address space is under-powered then processor reallocation may be necessary
  - solution?

- lazy address switching
  - the scheduler prefers to schedule threads from the same address space, until it really has to give processor to another address space
- client's processor given to the server via the kernel
- processor returned to the client by the server upon completion of the call via the kernel
- the thread management system in each address space does the voluntary load balancing

# Lazy Address Switching Detail

- Whenever a message is sent
  - the sender is de-scheduled from the processor
  - another thread from the sender's addr. space is scheduled on the same processor
  - if no such thread exists, the processor will run a special low-priority thread that looks for "underpowered" receivers:
    - underpowered: has msgs to receive, but is not scheduled on a processor
  - if an "underpowered" receiver exists, the processor is assigned to it

# What would this mean for uniprocessors?

- the client address space keeps sending requests (from different threads) and the receiver(s) will only be scheduled after all requests are sent!

# Example

- Client has an editor with two thread T1 and T2
- T1 invokes a procedure in a window manager, then upon return invokes a procedure in a file cache manager
- T2 invokes a procedure in a file cache manager
- Initially, the editor and window manager are running on two processors, the file cache manager is not scheduled on a processor.

- What is the sequence of events?

# URPC system

- two software packages (Figure 2)
  - fast threads for thread management
  - channel management and message primitives in the URPC layer
- solution bets on sufficient processor power in each address space
- client driven processor reallocation could be better than some fixed kernel policy

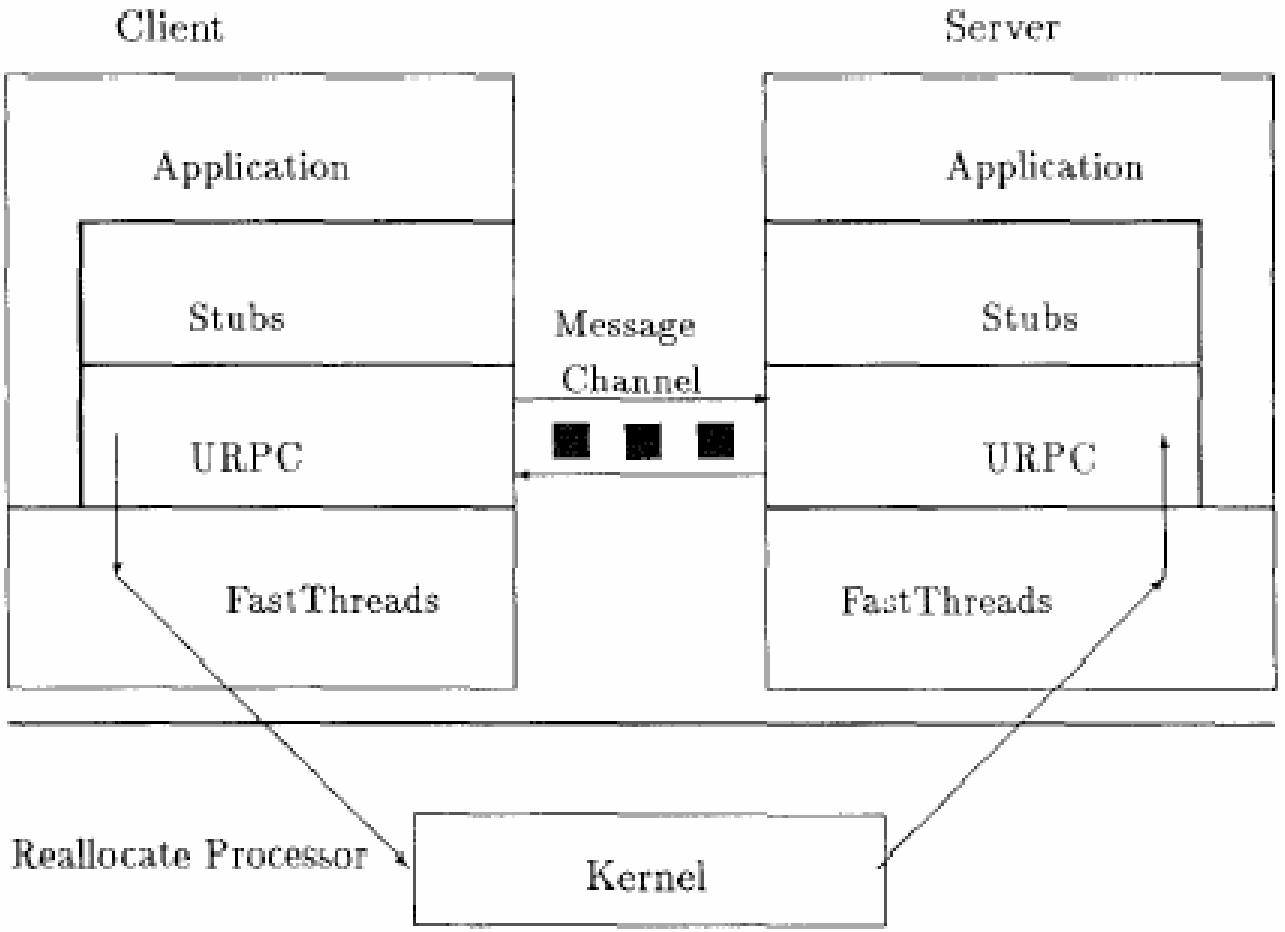



Fig. 2 The software components of URPC.

# URPC design decisions

- processor reallocation without any scheduling decision – amortize cost of kernel involvement over multiple cross-address space calls
- data transfer via shared memory
  - no kernel copying, no dynamic protection checking
  - message channels protected by polling style t&s locks
- user level threads
  - 100 times cheaper than kernel threads for context switches

# Critique

- The problem with lazy address switching is that the assumptions are optimistic:
  - it assumes that a delay in processing a request will not degrade performance
  - it assumes that the server of an RPC call has or will soon have a processor to run on
- what about
  - single threaded applications
  - real-time application
  - high-latency I/O operations
  - high priority invocations
- A combination of lazy and “eager” address switching is necessary

# Performance

- will URPC always perform better than kernel-based RPC?
- P - number of processor
- T - number of threads
- if T > P then URPC may be worse, why?
- T <= P then, no processor reallocation and URPC will be better

# Kernel based IPC

- cost of switching address spaces for sync. and comm.
- mismatch in design of thread packages
  - thread management at user level

  => high performance
  - communication amongst threads via kernel

  => low performance
- solution?
  - pull IPC out of the kernel

- Focus in paper:
  - control transfer during IPC
    - describes lazy; alternative eager (in LRPC, doors…)
- What about actual data transfer, what are the options where?
  - shared memory, plus sync. and ‘protocol’
  - message passing:
    - the usual, copy in/out of kernel
    - what about small data (e.g., on some archs, with eager ctx switch)
    - what about bulk data
  - Local Procedure Calls in XP