How can a technology be harmful?

- A good technology should
  - Encourage sound software design
  - Lead to robust, bug-free, high performance implementations
  - Be well supported
  - Co-exist well with other important technology options
A famous example

- Programming languages always support the “goto” statement
- In the early days, goto was common.
  - In BASIC, goto was elevated to a role of primacy for control and flow
  - Fortran also uses goto extensively
- But goto can be used well or used badly…

Misuse of goto?

- With goto statements
  - One can easily end up with spaghetti
  - It isn’t obvious to the person reading code where control transfers could originate
  - Loss of “structure”
Goto considered harmful

- A classic paper, now mostly forgotten
  - Go To Statement Considered Harmful
    Edsger W. Dijkstra.
    Communications of the ACM, Vol. 11, No. 3, March 1968
  - “The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one’s program…”

RPC is at the center of the modern universe

- Remote procedure call is at the core of most modern architectures
  - “3-tier” database architecture
  - COM and DCOM in Microsoft Windows
  - SOAP in Web Services
  - RMI in J2EE and CORBA
  - … the list goes on and on
- Almost all the flaws seen with goto are also seen (in slightly different forms) with RPC!
The basic RPC protocol

- Client “binds” to server
- Server registers with name service

Client prepares, sends request to server.
Server receives request and provides service.

Client server
"binds" to server
prepares, sends request
receives request
The basic RPC protocol

- Client: "binds" to server, prepares, sends request
- Server: registers with name service, receives request, invokes handler, sends reply
The basic RPC protocol

- "binds" to server
- prepares, sends request
- unpacks reply

Compilation stage

- Server defines and "exports" a header file giving interfaces it supports and arguments expected. Uses "interface definition language" (IDL)
- Client includes this information
- Client invokes server procedures through "stubs"
  - provides interface identical to the server version
  - responsible for building the messages and interpreting the reply messages
  - passes arguments by value, never by reference
  - may limit total size of arguments, in bytes
Binding stage

- Occurs when client and server program first start execution
- Server registers its network address with name directory, perhaps with other information (e.g. might use UDDI)
- Client scans directory to find appropriate server
- Depending on how RPC protocol is implemented, may make a “connection” to the server, but this is not mandatory

Data in messages

- We say that data is “marshalled” into a message and “demarshalled” from it
- Representation needs to deal with byte ordering issues (big-endian versus little endian), strings (some CPUs require padding), alignment, etc
  - E.g. SOAP over XML
- Goal is to be as fast as possible on the most common architectures, yet must also be very general
RPC versus local procedure call

- Restrictions on argument sizes and types
- New error cases:
  - Bind operation failed
  - Request timed out
  - Argument “too large” can occur if, e.g., a table grows
- Doesn’t guarantee “exactly once semantics”
- Costs may be very high
- ... so RPC is actually not very transparent!

RPC costs in case of local destination process

- Often, the destination is right on the caller’s machine!
  - Caller builds message
  - Issues send system call, blocks, context switch
  - Message copied into kernel, then out to dest.
  - Dest is blocked... wake it up, context switch
  - Dest computes result
  - Entire sequence repeated in reverse direction
  - If scheduler is a process, context switch 6 times!
**RPC example**

- Source does `xyz(a, b, c)`
- Dest on same site
- O/S

**RPC in normal case**

- Source does `xyz(a, b, c)`
- Dest on same site
- O/S

*Destination and O/S are blocked*
RPC in normal case

Source, dest both block. O/S runs its scheduler, copies message from source output-queue to dest in-queue

Source does \texttt{xyz(a, b, c)}

Dest on same site

O/S

Dest runs, copies in message

Dest on same site

Source does \texttt{xyz(a, b, c)}

O/S

Same sequence needed to return results
Important optimizations: LRPC

- Lightweight RPC (LRPC): for case of sender, dest on same machine (Bershad et al.)
- Uses memory mapping to pass data
- Reuses same kernel thread to reduce context switching costs (user suspends and server wakes up on same kernel thread or “stack”)
- Single system call: send_rcv or rcv_send
LRPC

Source does  
xyz(a, b, c)

Dest on same site

Control passes directly to dest

arguments directly visible through remapped memory

LRPC performance impact

- On same platform, offers about a 10-fold improvement over a hand-optimized RPC implementation
- Does two memory remappings, no context switch
- Runs about 50 times faster than standard RPC by same vendor (at the time of the research)
- Semantics stronger: easy to ensure exactly once
U/Net

- Low latency/high performance communication for ATM on normal UNIX machines, later extended to fast Ethernet
- Developed by Von Eicken, Vogels and others at Cornell (1995)
- Idea is that application and ATM controller share memory-mapped region. I/O done by adding messages to queue or reading from queue
- Latency 50-fold reduced relative to UNIX, throughput 10-fold better for small messages!

U/Net concepts

- Normally, data flows through the O/S to the driver, then is handed to the device controller
- In U/Net the device controller sees the data directly in shared memory region
- Normal architecture gets protection from trust in kernel
- U/Net gets protection using a form of cooperation between controller and device driver
U/Net implementation

- Reprogram ATM controller to understand special data structures in memory-mapped region
- Rebuild ATM device driver to match this model
- Pin shared memory pages, leave mapped into I/O DMA map
- Disable memory caching for these pages (else changes won't be visible to ATM)

U-Net Architecture

User’s address space has a direct-mapped communication region

ATM device controller sees whole region and can transfer directly in and out of it

... organized as an in-queue, out-queue, freelist
U-Net protection guarantees

- No user can see contents of any other user’s mapped I/O region (U-Net controller sees whole region but not the user programs)
- Driver mediates to create “channels”, user can only communicate over channels it owns
- U-Net controller uses channel code on incoming/outgoing packets to rapidly find the region in which to store them

U-Net reliability guarantees

- With space available, has the same properties as the underlying ATM (which should be nearly 100% reliable)
- When queues fill up, will lose packets
- Also loses packets if the channel information is corrupted, etc
Minimum U/Net costs?

- Build message in a preallocated buffer in the shared region
- Enqueue descriptor on “out queue”
- ATM immediately notices and sends it
- Remote machine was polling the “in queue”
- ATM builds descriptor for incoming message
- Application sees it immediately: 35usecs latency

Protocols over U/Net

- Von Eicken, Vogels support IP, UDP, TCP over U/Net
- These versions run the TCP stack in user space!
- Later in course will look at other complex protocols over U/Net
VIA and Winsock Direct

- Windows consortium (MSFT, Intel, others) commercialized U/Net:
  - *Virtual Interface Architecture (VIA)*
  - Runs in NT Clusters
- But most applications run over UNIX-style sockets (“Winsock” interface in NT)
- Winsock direct automatically senses and uses VIA where available
- Rather complex, somewhat fragile

The “way of RPC”

- RPC typically seeks *transparency*
  - Goal is to make remote procedure calls look as much like local ones as possible
  - User codes a remote invocation as if it was a call to an object in her own program
- Is network transparency a good thing?
The reality?

- RPC promotes an illusory form of locality
  - Local procedures take arguments by value and by reference. RPC arguments must be passed by value – copied into messages
  - Invoking a local procedure is cheap. Invoking a remote procedure can be very costly
  - A local procedure can’t fail mysteriously. A remote invocation can fail without clarity as to the outcome

Half-way transparency

- Developer necessarily does know that some objects are local and some are remote
  - She codes accordingly...
  - But this means some of the logic of the program is “hidden”
  - The code can look quite simple yet the reasoning behind it may be very subtle
- Later, other people will have to maintain her program. They may not recognize these subtle considerations!
Performance issues

- RPC performance is very challenging
- With TCP we saw that key is to keep both the sender and the receiver active
  - The purpose of the TCP sliding window is to mask network latency
- With RPC, client and server tend to run in lock-step

Classical responses?

- Do nothing about this
  - Now either client or server is active but never both at once
  - Effect can be a very choppy execution
- Have many RPC’s concurrently
  - Client and server need multiple threads
- Or do RPC’s asynchronously
  - But errors can be very confusing!
- Examine each option in turn
Do nothing

- Execution will often be very slow
- Moreover, the larger the latency of the network, the bigger the problem!
  - For reading objects from a server, this forces the application to cache very aggressively
  - But now performance will vary depending on cache hit rate!
Multiple concurrent RPCs

- Idea is to somehow dispatch many requests in parallel
- This raises several issues
  - Because we are now using threads the programmer needs to program in a thread-safe manner
    - Worry about critical sections, reentrancy
    - Many will make mistakes!
  - Also worry about ordering issue
    - E.g., application does a, b, c but results come back c, a, b causing confusion

Multithreading debate

- Three major options:
  - Single-threaded server: only does one thing at a time, uses send/recv system calls and blocks while waiting
  - Multi-threaded server: internally concurrent, each request spawns a new thread to handle it
  - Upcalls: event dispatch loop does a procedure call for each incoming event, like for X11 or PC’s running Windows.
Single threading: drawbacks

- Applications can deadlock if a request cycle forms: I’m waiting for you and you send me a request, which I can’t handle.
- Much of system may be idle waiting for replies to pending requests.
- Harder to implement RPC protocol itself (need to use a timer interrupt to trigger acks, retransmission, which is awkward).

Multithreading

- Idea is to support internal concurrency as if each process was really multiple processes that share one address space.
- Thread scheduler uses timer interrupts and context switching to mimic a physical multiprocessor using the smaller number of CPU’s actually available.
Multithreaded RPC

- Each incoming request is handled by spawning a new thread
- Designer must implement appropriate mutual exclusion to guard against “race conditions” and other concurrency problems
- Ideally, server is more active because it can process new requests while waiting for its own RPC’s to complete on other pending requests

Negatives to multithreading

- Users may have little experience with concurrency and will then make mistakes
- Concurrency bugs are very hard to find due to non-reproducible scheduling orders
- Reentrancy can come as an undesired surprise
- Threads need stacks hence consumption of memory can be very high
- Deadlock remains a risk, now associated with concurrency control
- Stacks for threads must be finite and can overflow, corrupting the address space
Threads: can spawn too many

Thread spawned, but blocks

event
Threads: can spawn too many

Eventually, application becomes bloated, begins to thrash. Performance drops and clients may think the server has failed

Ordering issues seen on multithreaded server

- incoming RPC’s could be out of order
  1. Move_cursor(100,100)
  2. Draw_box(10,10)
  3. Move_cursor(150, 75)
  4. Draw_line(15,20)

Order sensitive code!
Few programmers can handle ordering, concurrency!

- Programming languages are just poorly suited to concurrency
  - Too many “hidden considerations”
  - Thus concurrent code is often incorrect code
- Ordering issue is not directly tackled in threading packages, must be solved by developer
- And thus RPC forces a tradeoff
  - Slow but robust vs multithreaded but buggy!

Asynchronous invocations

- The idea here is to separate the request from the response
  - Request is merely “acked”
  - Response is via a “callback” RPC
- Ideally, result is a stream of requests
  - But what if a request mid-stream triggers an exception?
  - Similarly, what if client needs to change its execution plan?
Asynchronous streaming

- Move cursor(100,100)
- Draw_box(25,25)
- Move cursor(10,10)
- Draw_line(10,15)

Network/server failure handling – nail in the coffin

- RPC detects failure when invocations time out
- User can control timeout but
  - Timeout could be due to a slow server
  - Could be a network problem
    - Packet loss… congestion… lost connection
  - Could be a crashed server
Different applications may see inconsistent timeout!

- Timeout is a very arbitrary thing
- No agreement occurs or coordination
  - We could introduce a failure detector…
  - … but that would be a non-standard solution and architectures like Web Services lack a hook to do so

Suppose failure is real

- What next?
- We would ideally like to fail-over to a new server and reissue request
  - But how can we find out if prior request was finished?
  - RPC lacks a way to even identify requests in a standard manner!
Does WS TRANSACTION help?

- Recall that transactions let us say:
  - begin_xtn()
    - read_action() or write_action()…
    - read_action() or write_action()…
  - commit_xtn()
- Can we combine RPC with transactions?

Combining RPC and transactions

- Two issues arise
  - What if we do calls to multiple objects in a single transaction?
    - All need to participate in the commit/abort decision phase!
  - What if a failure occurs during a commit request?
    - Now client is back in the original situation: no idea what the outcome was!
2-phase commit

- The usual response to our first concern
- When client is ready to commit
  - Platform contacts each object
  - Asks “are you ready to commit”
  - Object must say “prepared”
  - If all are prepared, commit. Else abort

Preparing to commit

- Often must force information to a disk
  - Problem is that once prepared, even if the server crashes, on recovery it will still need to commit
  - Indeed, on recovery server must have a way to check outcomes…
- So this can be slow
2PC is quite slow

- A researcher named Skeen once proposed a 3-phase commit
  - Gives better availability
- IBM liked the work
  - Bruce Lindsey and C Mohan: “Cool”
  - Us: “Will you use it?”
  - Lindsey: “We would love to. But 2PC is already too slow for us!”

A sad sorry story

- Huge complexity
- Performance problems
- Inconsistency issues
- Platforms lack needed hooks
- Concurrent programming imposed on naïve and incompetent programmers
- Forced use of transactions (which raise issues of their own... subject for some other lecture!)
Why so sad?

- Web Services are RPC based
- So are major new platforms like .NET
- … and trends will only make this more and more extreme

- In the future, we’ll all be RPC programmers. And we’ll all experience all of these issues directly!