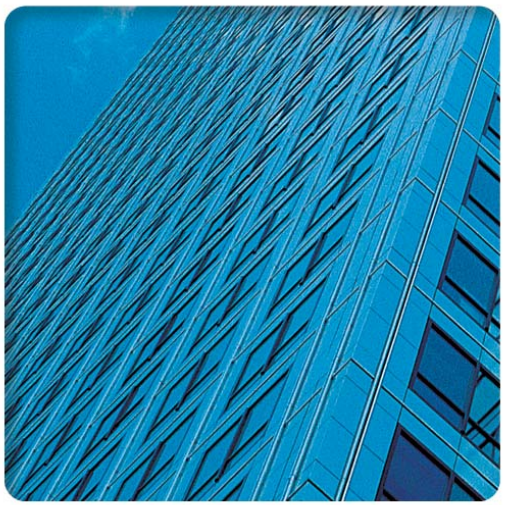


Symbian OS Architectural Paradigms

(or how these came about)

John Pagonis
Symbian Developer Network

Today's menu ;-)

- Welcome
- Why this talk
- History and philosophy
- Forces and Paradigms
- Developing for Symbian OS

Welcome :-)

- This talk is about OS and framework design
- For people interested in knowing the “Whys”
  - … behind Symbian OS idioms
  - … behind Symbian OS C++ dialect and paradigms
  - … mobile development
- For people that like to “hack” below the surface
- For people familiar with (Symbian) OS, frameworks, language architecture.
- Why things are the way they are…..
- “Interrupts, help unblocking’’

Why this talk

- Because Symbian OS is different
- Because people may be tempted not to spend time to appreciate it :-/
- To tell people beforehand …
- Because of it's mass adoption and deployment we run a risk of badly educated engineering…

- "Never criticise what you don't understand"

About architecture…

- Every architectural feat exists within some context, where it evolves under (competing many times) forces. That context and such forces present the architect with constraints and affordances.
- For us ‘’small mobile computers’’ has always been that context.
- We’ve been operating in this context for the past 20+ years or so !
- Like any building that has stood there through time, forces around it may or will change. Fortunately for us software is a bit more adaptable and malleable…

Some historical background

- Early 80s: Spectrum ZX/81 Flight Simulator and other games
- 80s : 8bit Organiser circa ‘84 (I, II, P350, CM, LZ64 etc) and the Sinclair QL app suite – early frameworks
- Late 80s : SIBO/Epoc16 circa ’88 – WIMP(MC), HWIM(s3), XWIM(s3a) - emerging frameworks
- 90s : Epoc32 – EKA1, HCIL, Eikon – frameworks
- Late 90s – today: Symbian OS – EKA1/EKA2, Uikon, UIQ, S60, FOMA UI etc

Evolution…

- Games (efficiency and hardware insight)
- Vertically integrated products (hardware and software) and app development
- Product diversification, Frameworks, OS creation and evolution (many of)
- Software reuse and platformisation

- Z80 assembler
- C, systems thinking, UI, OPL, first object-based designs with C, limited comms, early OOP
- OOP/OOD, C++, Java, OPL, comms
- J2ME, Python, Perl, C++, many GUI frameworks, lots of comms, open to anything and anyone really…

Force: User Interaction and Responsiveness

- Designed for ordinary people to use and interact with, as opposed to most if not all other embedded and RT OSs out there.
- UI needs to be easy and responsive
- Lots of user initiated I/O
- Can’t waste CPU cycles, we need them for the user
- Many applications use many services and resources
- A-synchronicity is everywhere
- Framework needs to support the developer to manage complexity

Force: Battery Powered

- Designed for people to carry at all times (back in the 80s:-)
- At most times the system is waiting for the user
- Can’t waste battery by polling for inputs, timers etc
- System needs to cleverly manage hardware and switch-off whatever silicon is not needed at times
- But needs to wake up rapidly and interact with the user

Force: ‘’Always-on’’ , always pocket-able

- Designed to always be there for the user, since it is battery powered it is therefore mobile.
- Instant-on, ROM XIP operation
- Implications for UI design :Uncluttered, snappy, protect the user from bad choices
- Devices used on the go, under awkward situations, with one hand perhaps while doing other things
- Consistency is key, don’t surprise the user !
- Apps had to be built around a framework that honoured and enforced these requirements – OS too

Force: Reliability

- Designed to always be there and trusted by the user
- Dependable
- Can’t afford to lose user data
- Should not need reboots of course !!!
- Cannot waste resources → DoS
- Extra care in the framework to assist and remind the developer about this :-)
- Memory protection
- Fault containment

Force: Resource constrained

- Designed to **expect** resources to run out at the worst possible of times - and there wasn't much of them anyway!
- e.g., battery pulled out, out of RAM, memory card removal, comms failing
- All the system had to be developed in order to support this **mentality**, from Kernel to Apps.
- Thus the frameworks had to make sure developers will **care** about this and could not easily get away from thinking about it.

Force: Openness…

- Designed as an open system, where complex software can be build for it.
- Openness brings innovation, diversity and lots of opinions !
- Openness has complications and complexity.
- System has to be able to protect itself and the user and remind the developer about it !
- System has to assist developers to develop, re-use and extend……

Some paradigms…

In order to support the user-centric mentality and operate within the mentioned forces we used some design paradigms and idioms such as :

- Multithreading and pre-emptive multitasking
- Lightweight Micro Kernel OS design
- Client-Server, session based IPC
- Asynchronous services, Active Objects
- Cleanup Stack, Leaves, Traps for exception handling
- Re-usable frameworks for apps, middleware, GUIs
- …and descriptors

Language selection… C++

- C++ is a multi-paradigm programming language
- Probably not the best language for loosely coupled open world systems
- But we pretty much knew what the frameworks and apps would need and should be like
- It supports OOP
- Good for low level OS construction, strongly typed and works nicely with assembler.
- Has too many features and demands proper attention, only got standardised recently though…

Standard C++ ???

Late standardisation meant :

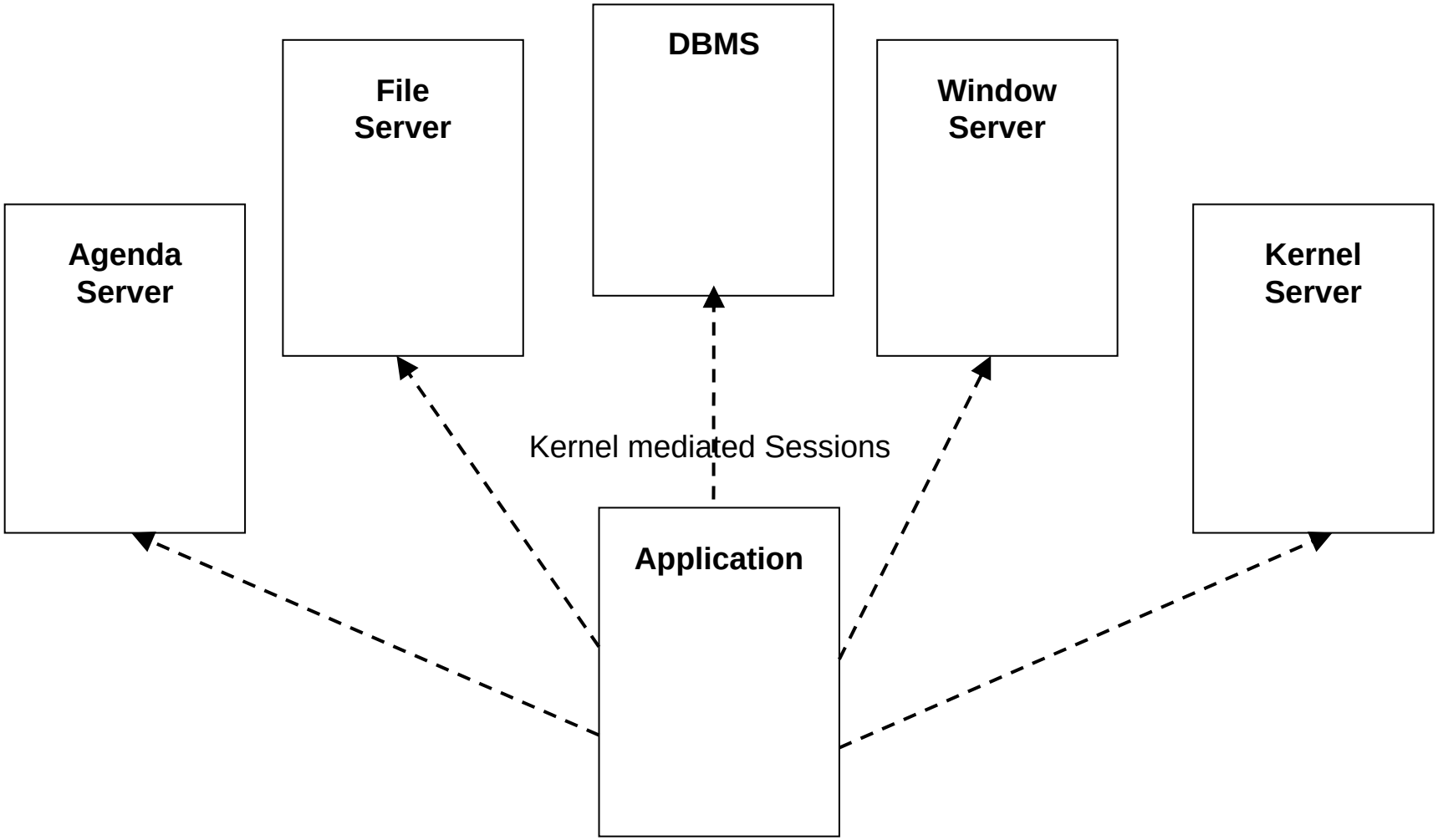
- We had to came up with our own exception mechanisms
- Came up with our own collections, container and buffer/string handling frameworks
- Concurrency support was not in the package

- …and we have been criticised ever since for designing something different that works and is domain specific :-(

Microkernel design – EKA1

- Lightweight Microkernel with client server architecture
- System servers as well as user servers run in user space
- Kernel uses Virtual Memory Model and MMU for memory protection
- Driver code runs kernel space
- Effectively one IPC mechanism – Client/server sessions
- Servers mediate access to shared resources and services
- Kernel deals with memory allocs and IPCs

Asynchronous services



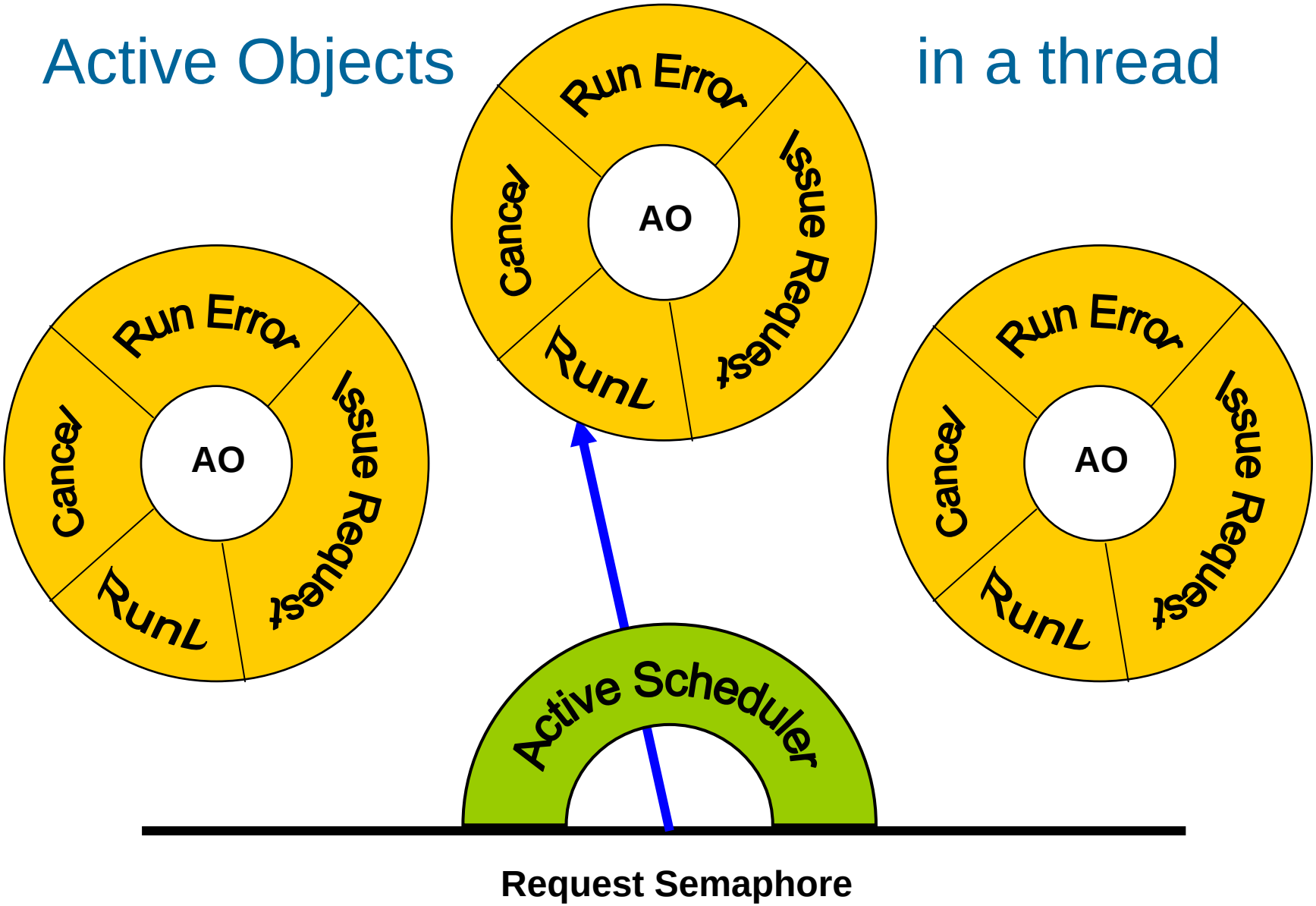
Many asynchronous services….

- But the OOP model is **serial** and so is vanilla C++
- Concurrency support was needed for asynchronous event handling and the multitude of client-server comms
- A lightweight model that introduces concurrency and **avoids synchronisation issues** is that of Active Objects
- Grady Booch talks about AOs in his 1990 book on ‘’Object Oriented Design with Applications’’. Defined as objects having their own thread of control.
- We used collaborative AOs within a thread instead

Active Objects

- Thus every thread got a *request semaphore*
- Most threads and certainly all apps and servers got an *Active Scheduler*
- So that it can schedule…. Active Objects
- Thus Active Objects became a lightweight mechanism to **encapsulate concurrent transactions** over session based IPC
- Simple: Issue a request, run when the request completes, if an error occurs handle the error.
- … priorities, run to completion, no pre-emption

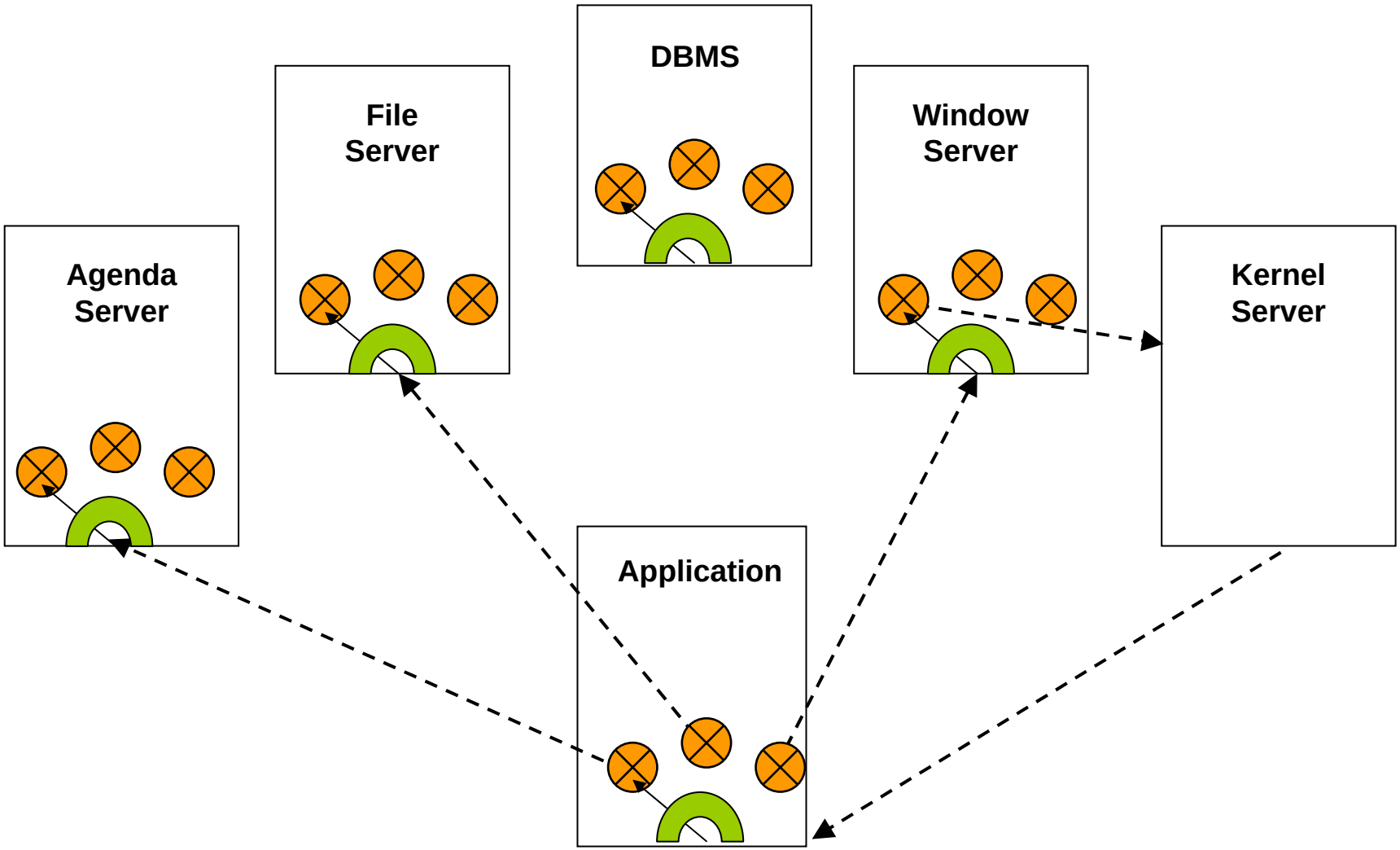
Active Objects in a thread



Active Object Transaction Encapsulation

- Use or open a session to some service provider (server)
- Add AO to the Active Scheduler
- Issue a request to that service that may complete asynchronously
- Set the AO active, thus notify the scheduler
- … handle the completion of the request in the `RunL`
- If there is an unhandled exception `RunError`
- ..or let the Active Scheduler know

Active Objects almost everywhere



Good things about Active Objects

- AOs help to manage complexity and synchronisation issues with concurrency within a thread without too much to worry about
- AOs encapsulate transactions and their error handling
- AOs help us avoid multi-threading for async event handling where it is not always needed or when it is more complicated or just an overkill
- AOs let servers manage many clients and many transactions with just one thread

..and the bad

- BUT they are **no panacea** and should not be used where they don’t fit the paradigm !!!
- Multi-threading instead of AOs is valid and must be used where it must !

About exception handling…

```
SomeMethodL()
{
tmpObject = new (ELeave) CHeapyObject()
CleanupStack::PushL(tmpObject)
DoSomethingThatMayLeaveL()
CleanupStack::Pop(tmpObject);
// …and pass ownership of that object
//OR possibly
CleanupStack::PopAndDestroy(tmpObject)
}
```

And why is that ?

- Poor or no support for C++ exception handling in compilers at the time
- CleanupStack is in your face → a good thing
- Compilers behind the scenes write all the exception handling code for you, they tend to be really conservative and churn much more code than a developer would
- ….this may change with newer compilers

How does the CleanupStack work ?

- It stores pointers to objects to be destroyed in case of an exception (a.k.a. Leave)
- These pointers are stored in nested levels
- Such levels are marked by the TRAP macro (think _almost_ of ‘C’ setjump/longjump here with exec call)
- When a Leave occurs, it makes sure it calls all d’tors (and cleanup items) of objects belonging to the corresponding Trap level.
- And the stack unwinds to the point of TRAP, returning an exception error code.

Trapping and Leaving

```
Cleanupstack::PushL(something);/* will stay on
   the CleanupStack if it Leaves below */

TRAPD(err,SomeMethodL());

if (KErrNone==err)

   {//do something }

else

   {//start handling the exception

   //and possibly propagate it by another  //Leave

   }

….possibly pop something
```

Architectural Evolution – EKA2

- New multi-threaded, pre-emptible RTOS Kernel
- In-fact is a Symbian OS personality on top of a Nanokernel – **many** personalities can co-exist ;-)
- O(1) scheduler
- System calls are pre-emptible as well, dual stacks
- Deterministic ISR, thread response, latencies etc
- Memory models and local allocator strategies can be plugged-in
- Many more IPC/ITC mechanisms such as local/global message queues, publish-subscribe, global anonymous queues, shared I/O buffers

More IPCs ?!!

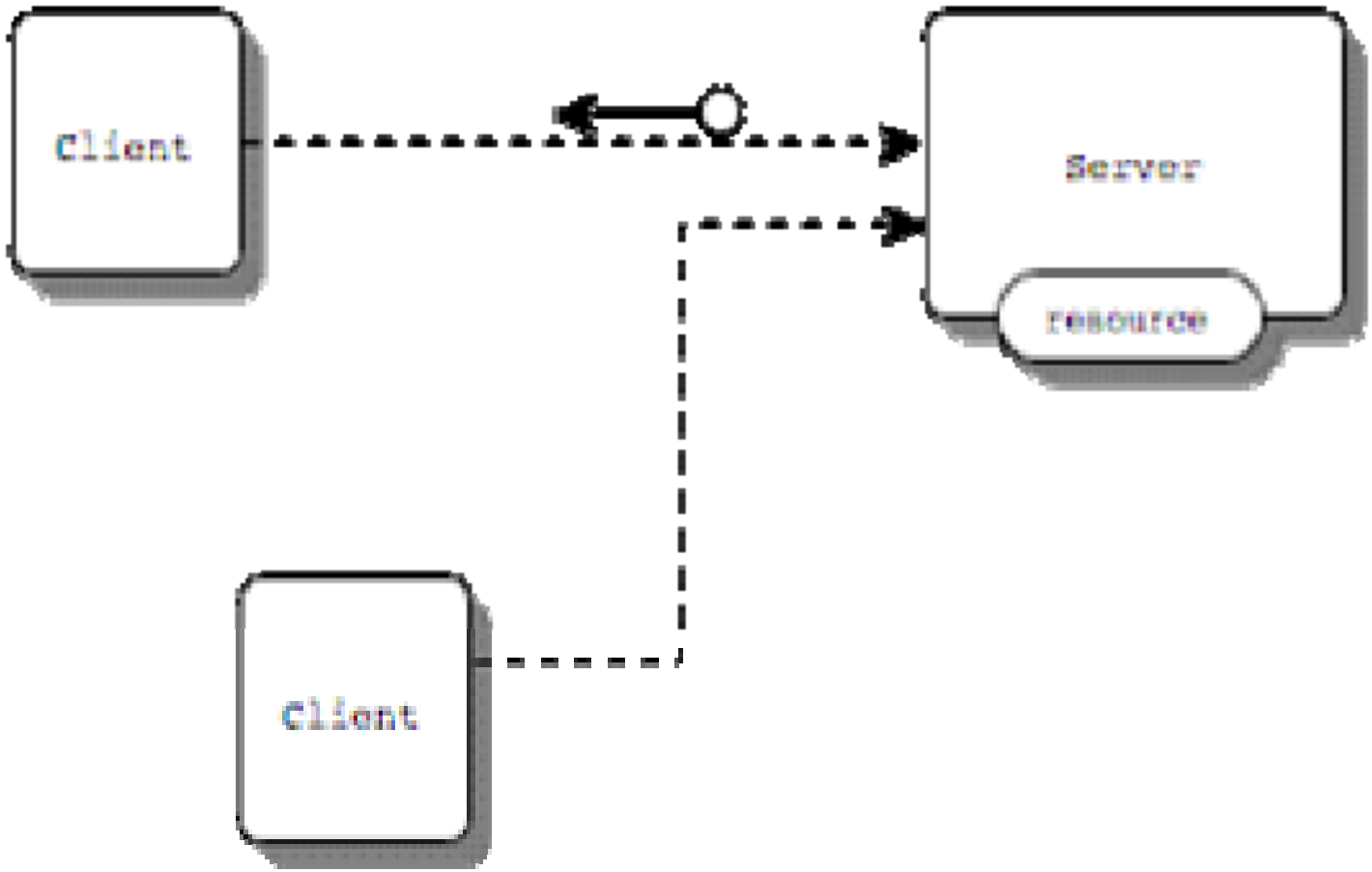
A major event in any OS's evolution is that of IPC addition. Let alone 3 of them !!

- ✓ Publish & Subscribe
- ✓ Message Queues
- ✓ Shared Buffer I/O – between driver and user space

Beyond Client-Server IPC

- Connection-oriented
- Client initiates request, server responds
- Guaranteed delivery and request completion mechanism
- Connection set-up and teardown is always synchronous in EKA1
- Good paradigm for when many clients needs to reliably access a service or shared resource concurrently
- Relation is one-to-one but not peer-to-peer

Beyond Client-Server IPC



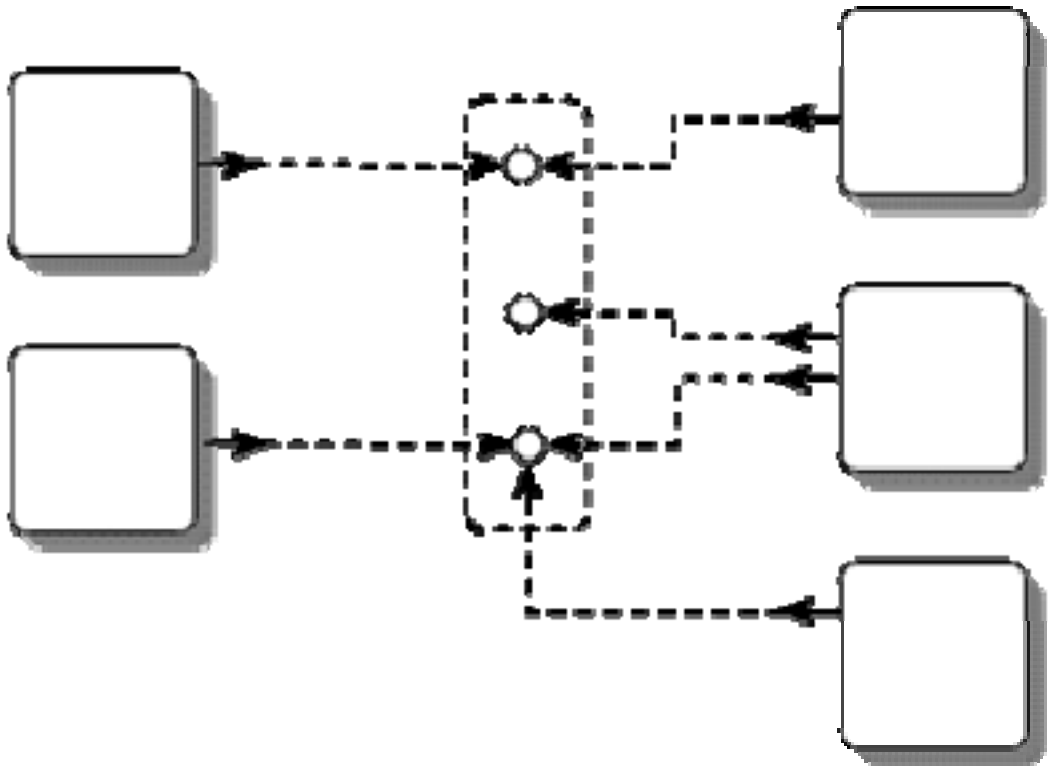
Client-Server IPC Limitations

- Not all I/O is user-initiated – system has evolved …
- Clients must know which server provides the service they want
- It requires that permanent sessions between client and server is maintained
- Deadlock potential between servers due to synchronicity of session creation/teardown
- Not really suitable for event multitasking
- Although delivery is guaranteed, there is no RT deterministic guarantee of message delivery.

Publish & Subscribe (a.k.a. Properties)

- Define and publish system-wide properties
- Properties are communicated to *many peers* asynchronously
- Both user and kernel side (via similar APIs)
- Properties are single data values, uniquely identified by an integral key
- Properties have *identity* and *type*
- The identity and type are the only things that need to be shared between publisher(s) and subscriber(s)

Publish & Subscribe



Publish & Subscribe operations

- Define: Create a property and define type and identity
- Delete: Remove a property from the system
- Publish: Change the value of a property
- Retrieve: Get the current value of a property
- Subscribe: Register for notification of changes
- Unsubscribe: Deregister for notification of changes

Can be used transiently or by prior ‘attachment’

…attachment helps to do RT deterministic operations in EKA2

Publish & Subscribe characteristics

- Definition and deletion coupled in the same thread
- Either publisher or subscriber may define a property !!
- Connection-less paradigm (between P – S)
- Publishers and subscribers don’t need to know about each other or link to special client APIs etc.
- One to many and many to many communication
- Attached or transient operation
- Properties are read and written atomically
- Registration, change, notification, retrieval

Message Queues

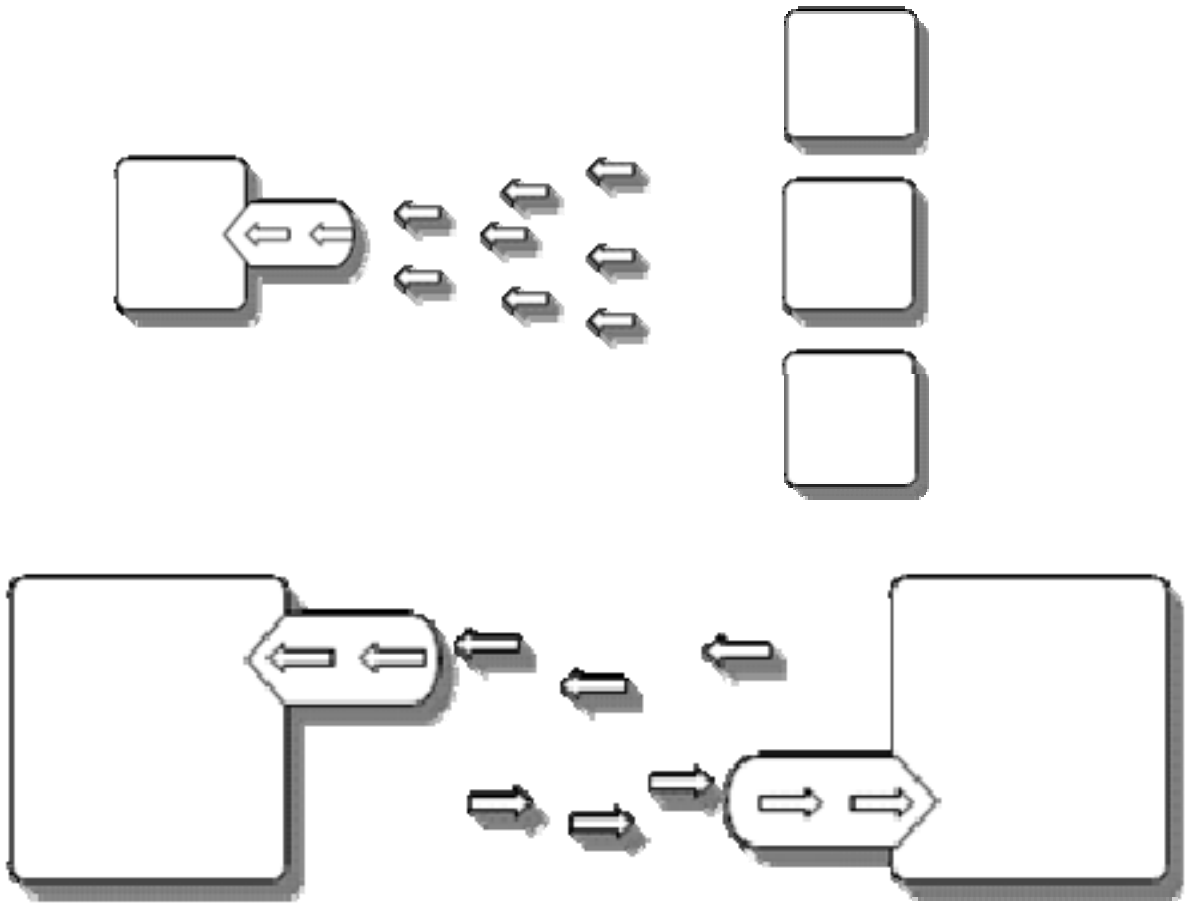
- Peer-to-peer
- Many-to-many
- Fire-and-forget communication semantics
- Guaranteed delivery of messages to queues
- ..but final delivery to reader isn't
- Queues are dimensioned at the point of creation, messages are short and fixed size
- Lowest overhead EKA2
- deterministic IPC mechanism

Message Queues operations

There are five basic operations that can be performed on a message queue:

- Creating/opening a message queue
- Sending a message
- Receiving a message
- Waiting for space to become available in the queue
- Waiting for data to arrive in the queue

Message Queues data flow



Message Queues characteristics

- Kernel managed objects
- Queues may run out of space
- Senders can block on sending
- Senders can be notified of potential overflow thus retry
- Queues can be named and globally visible
- Queues can be process local only
- Queues can be anonymous and globally accessible in EKA2
- Queues may have multiple readers (with health warnings)
- Neither messages nor queues are persistent
- Symbian OS Queues can be *typed*
- *Message structures are <256 bytes*

Getting to the new IPCs

You can get to P&S and Message Queues API by using any C++ SDK based on v8.0 or higher

Nokia Series 60 2nd ed FP2

Can't get to EKA2-only features yet

They have been back ported to latest 7.0s releases

……like Series 60 2nd edition FP1 :-)

Thank you !

Q & A

“Symbian OS Explained”, by Jo Stichbury

“Symbian OS for Mobile Phones vol2” , by R. Harrison

..and of course visit the Symbian Developer Network