Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Streams, Signals and Binding (Presentation to BNR)

Dave Otway

Abstract

Presentation to BNR on 18th January 1995

APM.1396.01 **Approved** 17th January 1995
Briefing Note

Distribution:
Supersedes:
Superseded by:

Streams, Signals and Binding

Dave Otway

djo@ansa.co.uk

Motivation

- endpoints of high bandwidth data streams used to be hardware
  - mips now available to process them in software
  - new media and formatting standards -> soft engineering
- multi-media increasingly used in mainstream applications
- integration of business functions requires real-time systems to interwork with non real-time systems
- synchronous programming research enables synchronous, predictable programs to be executed in a basically asynchronous system

Data Flows

- data flows are unidirectional from source(s) to sink(s)
- each endpoint may or may not interface to a software object
- the source(s) of a data flow must be bound to its sink(s) before communication can take place
- no correlation between sources / sinks and clients / servers
- combining multiple data flows into a bidirectional stream provides:
  - a simple binding reference for related flows
  - easy exploitation of duplex communications engineering and technology
  - abstraction of connection models

Framing

- asynchronous data flows have application defined frames
- some synchronous data flows have natural frames, others do not
- even when there is no natural framing, the programmer must decide on the appropriate size of data unit for each processing step
- even when the framing is arbitrary it is simpler, more efficient and more portable to do the framing in the engineering [just like stubs]
- all data flows can be regarded as a sequence of frames at the point where they interact with an application program

  [this does not imply anything about the framing characteristics of hardware endpoints]

Frame Formats

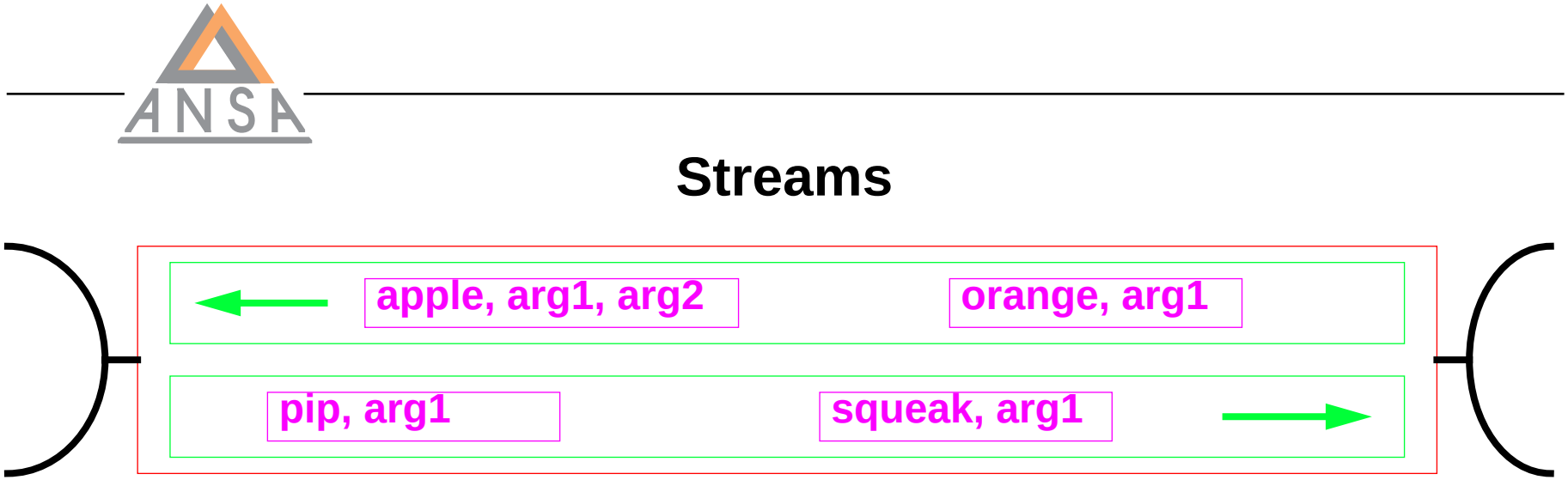
- multiple frame formats in the same flow enable:
  - frames which differ little from the previous one to be sent as a delta
  - the compression algorithm to be changed between frames
  - application level flow multiplexing / demultiplexing
  - in band control
- frames can be differentiated by names or numbers
  - numbers would preclude the use of multiplexing, in band control and conformance based type checking
- the data in a frame can be described by typed arguments
  - an argument is a binding to an (operational or stream) interface

Stream Signatures

- a stream signature that enables conformance based type checking is:

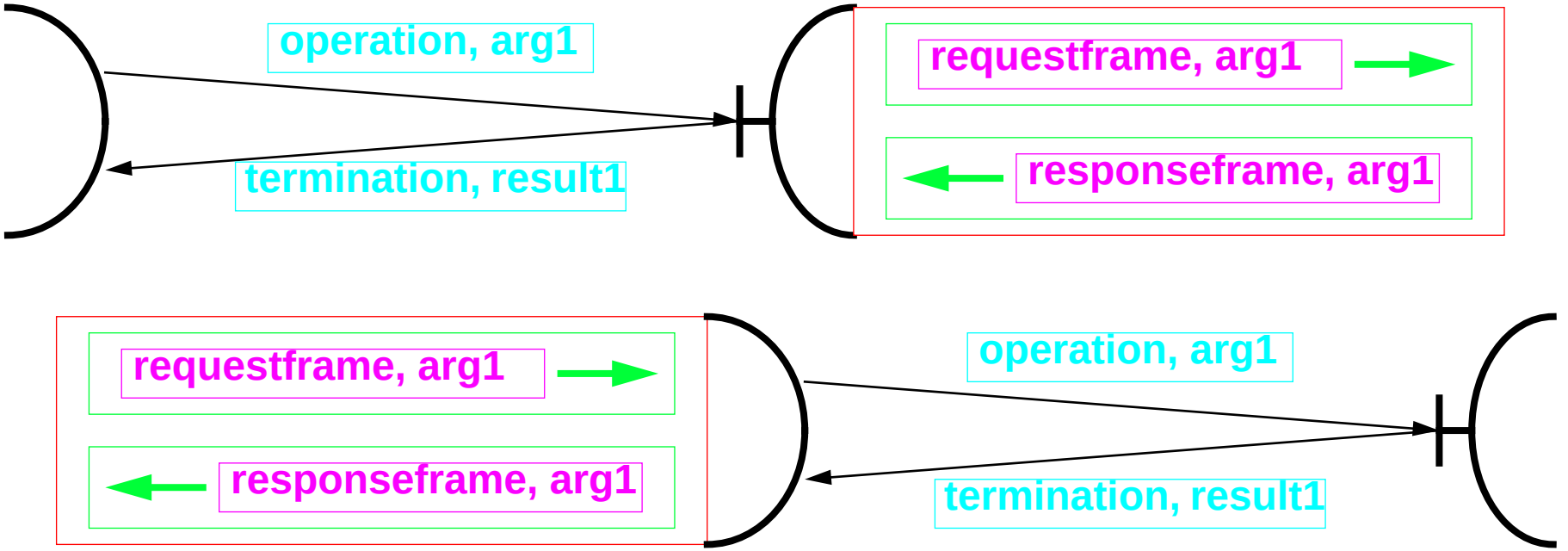
```
frame     =  frameName “(“ { typeExpression } “)”
direction =  “>>” | “<<“
flow      =  direction “(“ { frame } “)”
stream    =  “stream” “(“ { flow } “)”
```

- since frames are never dispatched
  - they don’t need bodies
  - their arguments don’t need names
- the same signature grammar can be used to describe both endpoints
  - with the direction of the flows reversed



Streams

- A stream has a set of flows
- A flow has a set of frames (or signals) and a direction
- A frame has a name and a set of typed arguments
- Streams are typed and can be conformance type checked
- Frames are transmitted by non-blocking writes and read by blocking reads



- **A stream interface can be programmed to use or provide an operational interface**
- **But relies on the programmer correctly implementing the operation invocation semantics**

Synchronous Programming

- a reactive system continuously interacts with its environment
  - its execution is divided into a sequence of discrete instants
  - each instant reacts to its inputs and produces the corresponding outputs
- the synchronous hypothesis states all reactions are instantaneous
  - simplifies reasoning by removing all concurrency between instants
  - execution of communicating threads in the same instant are serialised
- deterministic behaviour
  - bounded execution paths, calculable in advance
  - with guaranteed resources:
  - programs have predictable timing and reproducible behaviour
    [even in asynchronous systems]

Signals

- each input or output must be a discrete message so the program can determine which instant it must react to it or emit it
- each input or output is defined by a named signal with typed arguments
- each signal has a defined direction (in or out)
- signals are not broadcast, but grouped into interfaces
  - gives same local and remote semantics
  - enables bindings to be scoped
  - enables many instances of a signal interface to be used
- reactions are synchronised with real-time by input time signals

Transmission and reception of signals

- **transmitting signals is similar to invoking operations**

```
transmission = unit “!” signalName block
```

  - **which only blocks while the signal is queued - there is no reply**

- **receiving signals is done by an expression:**

```
reception = unit “?” signalName
```

  - **which blocks until the signal arrives and then returns the arguments**

- **examples:**

```
clock?hour ; bell!ring
[clock?second ; clock?second] || [button?press ; bell!ring]
```

Waiting and testing for signals

- the basic expression for testing for the presence of a set of signals is:

```
condition = "(" { unit "?" signalName } ")"
```

- a non blocking test to see if all of a set of signals are present is:

```
presence = "present" condition
```

- a blocking wait until all of a set of signals have been received is:

```
await = "await" condition
```

- the signals may be received in any order and in different instants
- signals are promoted to the instant in which the last signal is received

Watchdogs

- sometimes a sequence of reactions needs to be aborted
  - a watchdog expression will execute a block while watching a condition

    `watchdog = “during” block “watch” condition block`
- if the condition becomes true before the watched block terminates then its execution is aborted and the alternative block is executed
- if the watched block completes its execution then the watchdog expression terminates
- the condition semantics are the same as for await
  - all signals are promoted to the last instant
- if the (dynamically) last signal in the condition would also enable the completion of the watched block
  - the watched block is completed

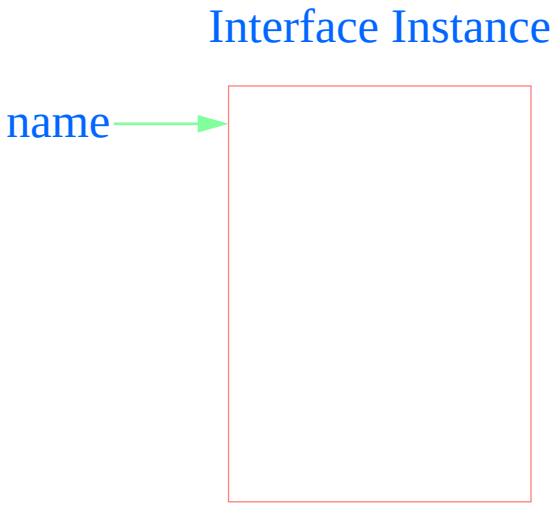
Summary of proposed extensions

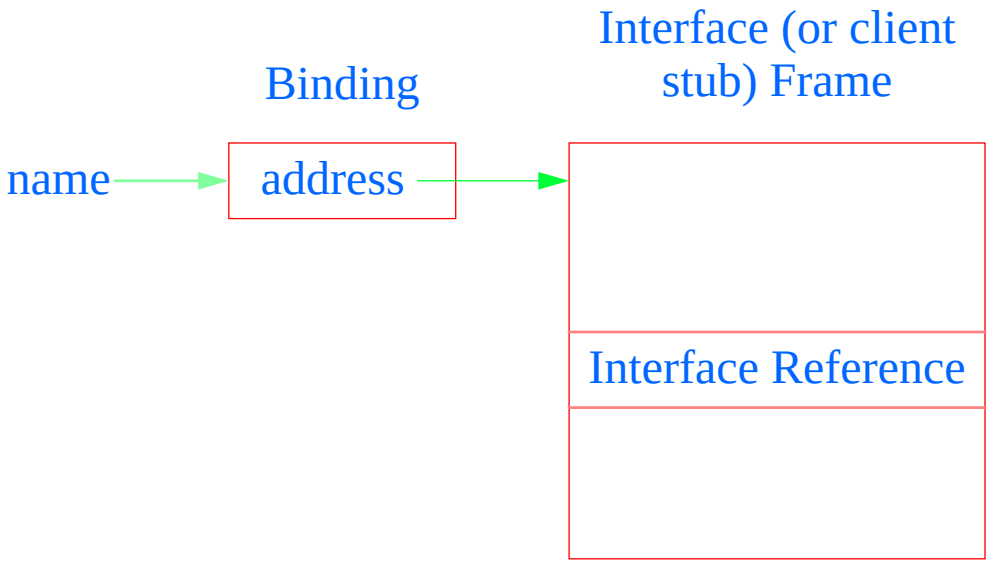
- **add the following new constructs**

```
signal       = signalName [attributelist]
                          “(“ { typeExpression } “)”
direction    = “>>” | “<<“
flow         = direction  [attributeList] “(“ { signal } “)”
stream       = “stream” [attributeList] “(“ {flow} “)”
transmission = unit “!” signalName block
reception    = unit “?” signalName
condition    = “(“ {reception} “)”
await        = “await” condition
watchdog     = “during” block “watch” condition block
presence     = “present” condition
```

- **extend expression and type to accommodate the new constructs**

Computational vs Engineering Bindings





Re-assignment is done by re-binding of the name

Binding of the computational name is fixed

Re-assignment is done by overwriting the local address in the engineering binding

Implicit Bindings

- not under programmer control
- engineered for efficient resource utilisation and maximum scaling
- communication resources bound as late as possible (first use)
- communication resources released whenever binding is dormant
- use multiplexing wherever possible
- asymmetric
  - clients know about servers they are not currently using
  - servers have no knowledge about potential clients
- no bounds on throughput, latency or jitter (just best effort)

Why do we need more control over Bindings?

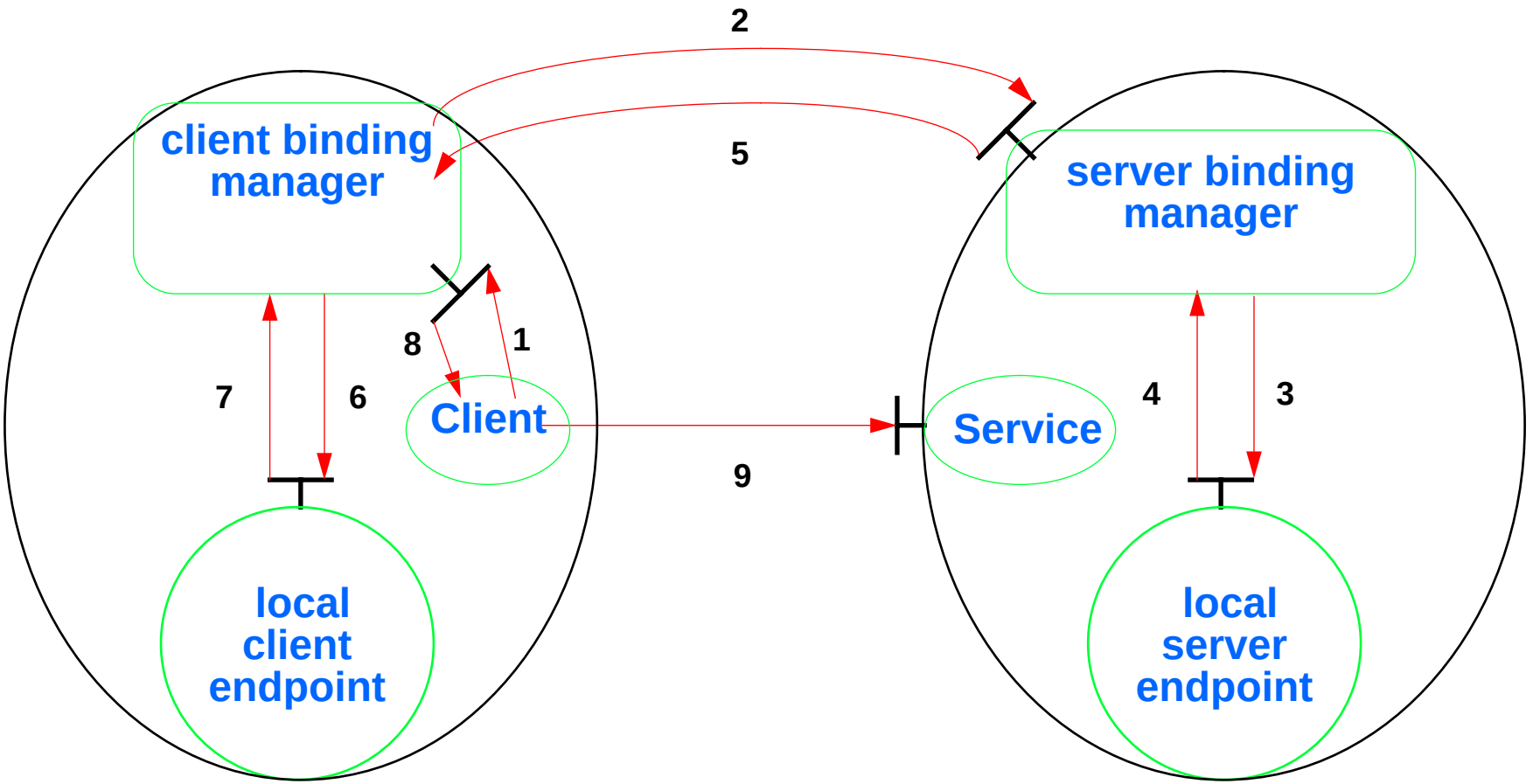
- to provide predictable communications
- to prioritise communications
- to provide non-operational stream communication
- to synchronise communication channels
- to provide a range of guarantees for:
  - synchronisation, ordering, dependability, security and performance
- to control the bind time
- to batch together a set of bindings
- to monitor the delivery of binding guarantees
- to manage bindings after bind time

Problem Areas

- **Security**
  - if an object hands over control of its bindings, how can it secure itself
  - if binding breaches encapsulation then how can secure systems be built
  - can you debug a system the size of a planet that is not self-protecting?
- **QoS negotiation and conformance checking**
  - is far too complex to build into the technology
  - is very difficult to build into the engineering
  - a small set of “standard” interoperable solutions would be very restrictive
  - QoS is dependent on resource management class (i.e. is open ended)
  - would benefit from the full range of distribution transparencies

Explicit Binding

- Generate a type specific local binder for each end of a binding
  - for client and server of an operational interface and both ends of a stream
- Perform all binding management by application program (libraries)
  - with the benefit of all distribution transparencies and tools
  - uses an implicit binding
- Each local binder is then requested to construct its end of the binding
- Provide type safe QoS specification, monitoring and control interfaces, multi-channel and multi-party bindings via matching attribute and engineering libraries
- Does not require management protocols for explicit binding to be provided in the communications technology


2
5
client binding manager
server binding manager
8
1
7
6
Client
Service
4
3
9
local client endpoint
local server endpoint

déjà vu

Back in the good old days when we were first wrestling with trading

A lot of people felt it should be part of the computational model

We eventually convinced them it was just another distributed application

It then became the dumping ground for system management functions

We had to drag each management function out one by one

"the management of binding is just another distributed application"