OPTIMIZED CORBA TO MEET SDR REQUIREMENTS

Dave Dohse and Eric Christensen
dave.dohse@gdds.com
July 15, 2004
Is CORBA Viable for an SDR?

CORBA is part of the JTRS SCA standard Operating Environment (OE) for Software Defined Radios (SDRs)
- Sometimes blamed for imposing processing requirements as much as five times that of legacy radio applications
- Inefficiencies in reality result from “less than optimal” deployment

Existing Navy DMR is an SDR with CORBA and POSIX
- Just 218 MIPS of combined Modem, Controller and I/O processing in DSPs and GPPs for UHF SATCOM Waveform
- 20% of Vendor B projection for implementation
- If DMR were made SCA compliant, no projected increase in MIPS (CORBA and POSIX already present)**

** Waveform application instantiation has potential to be slower under SCA because of Factory operations and XML parsing

MIPs Comparison for CORBA-based UHF SATCOM Waveform in an SDR

Operational SATCOM on a DMR Radio

- Projected Overall
- Modem DSP
- Modem GPP
- I/O (Voice and Data)

* September 2003 Military Radios Conference
Approaches to Optimizing CORBA

- Achieve optimized performance through judicious use of CORBA that preserves application portability and inter-ORB compatibility
  - Use of alternate data transports to TCP/IP optimized for embedded bus architectures
  - Minimize CORBA marshaling
  - Use persistent connections and CORBA one-ways in the communication datapath
    - Avoids latencies for connection setup and blocking for pending acknowledgements
    - No sacrifices for reliable message delivery that is more appropriately the responsibility of the transport layer
  - RT CORBA policies that assure predictable performance
    - Priorities, Mutexes, Threadpools

- The DMR radio supports 1000 connections and in excess of 500 threads for four simultaneously operating waveforms in a 233 MHz PPC750 processor
Incorporation of CORBA into an SDR-Based Application

![Diagram showing the integration of CORBA into an SDR-based application](image-url)
Efficient Application of CORBA to an SDR Begins with the Transport Layer

Use of stack for local calls avoids marshaling

Use pluggable transports instead of TCP/IP

- Optimized for embedded bus architectures (cPCI, LVDS)
- Extensible Transport Framework (ETF) provides common definition CORBA/Transport interface
Relation of CORBA Performance to Messaging Characteristics

Range of performance depends upon how CORBA is used

- Heavyweight messaging case involves TCP/IP and marshaling with a C++ ORB
- Stages of savings introduced by switching to cPCI transport, “C” implementation, 1-way calls and invocation on the stack
- RTOS task dispatch latency is shown for baseline comparison
Relation of CORBA Performance to Communication Messaging (2)

Examination of relative latency contributions offers insight to greatest opportunities for optimization

- 83% of the processing is attributed to excess TCP/IP overhead and “C” vs. C++ efficiencies†
- 9% 2-way callback vs 1-way overhead ‡
- 7% for marshaling and non-TCP/IP transport

† Derived by subtracting all other latency contributions from the overall latency of a C++ and TCP/IP-based 2-way call
‡ Data represents best-case number if blocking is not propagated beyond the first servant in a processing chain

* Latency (instruction cycles) = Processor Rate (MHz) * Data Size (bytes)/Throughput (Kbytes/Sec) where Processor Rate = 233 MHz and Data Size is either 30, 32 or 40 consistent with available measurements
Impact of Cascaded Blocking in 2-Way

1-way SYNC_WITH_TRANSPORT scope
- Client “blocks” until transport layer acknowledgement
- Nearly the same effect as SYNC_NONE scope with sliding window transport with window = 1
- Latency = “X” msecs (connection setup, marshaling, transport latency)

2-way SYNC_WITH_TARGET scope
- Client “blocks” until Target has completed execution at all servants – the diagram illustrates blocking propagated across 3 servants
- Transport acknowledgements do not propagate to client
- Latency = 3 degrees of propagation * (X + α) msecs where “X” is the 1-way SYNC_WITH_TRANSPORT latency, “α” is additional servant processing including marshaling of response parameters
Elements of Real-Time CORBA Assure Deterministic SDR Operation

- **Priority Mappings [CORBA 24.12]**
  - Normalizes priority of clients across RTOS’s to insure against priority inversion

- **Mutex Interface [CORBA 24.16]**
  - Extends RTOS Mutex protections to CORBA as a means of locking access to CORBA implementations (e.g., memory)

- **Threadpools [CORBA 24.17]**
  - Processing “lanes” (collections of execution threads) providing protection from encroachment by other threadpools

- **PrivateConnectionPolicy [CORBA 24.20]**
  - Allows the establishment of dedicated transport connections

- **Invocation Timeout [CORBA 24.21]**
  - Provides a uniform method of specifying timeouts relative to connection failures and delays prior to attempting renewal of connections
Is Fault Tolerant CORBA Suitable for an SDR?

High availability is desirable in SDRs
- CORBA failover mechanisms offer an opportunity to increase availability
- 3 possible approaches
  - Integrated within ORB
  - CORBA Service
  - Transparent Interceptor

Concerns
- Overhead of Multicast protocols (e.g., Totem) may be too great
- Can replicants sufficiently track primary objects (e.g., info hidden by local invocations)?
Conclusions

● When not used efficiently, CORBA can become a burden to an SDR

● Key to the effective use of CORBA in an SDR is the blending of minimum CORBA with optimized Transport layers, Pre-connects and Real-Time processing mechanisms

● General Dynamics has demonstrated that CORBA can be effectively applied to a production SDR
References


