JPL

Message Transfer Service for Space Applications

Overview, API & Current Status

Peter Shames, NASA/JPL

28 October 2002

Spacecraft On-Board InterFaces (SOIF) Background

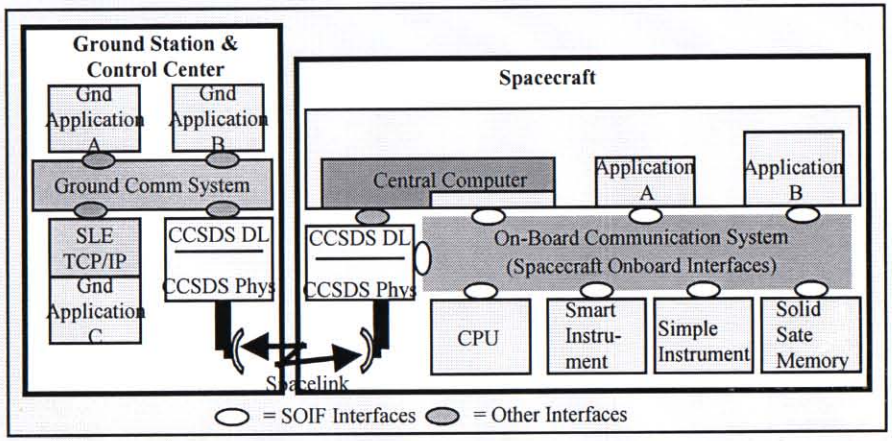
- CCSDS is an international organization that develops space data systems & communications standards
- SOIF (Panel 1K) began in late 1999, to develop a reference architecture and standards for spacecraft on-board interfaces
  - Physical level thru applications
  - Defining capabilities and services
- Message Transfer Service has been defined by a SIG within Panel 1K

SOIF Drivers & Enablers

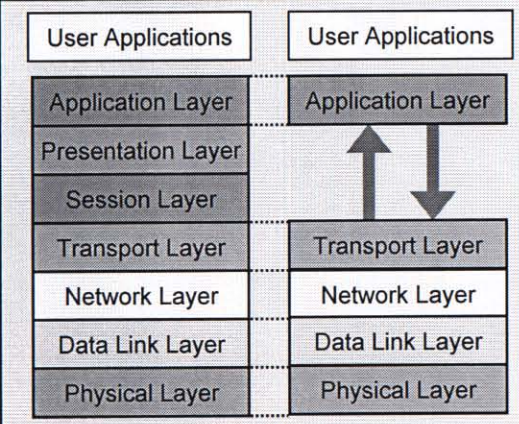
- Multi-processor and multi-S/C mission designs
- Complex distributed S/C hardware & software systems
- Powerful on-board processors, networks, and large scale data storage elements
- RT operating systems used in flight
- Requirements for cost and risk reduction
- Requirements for mass reduction

SOIF Context

The Context of SOIF: Flight and Ground



- The context of the Spacecraft Onboard Interface (SOIF) is related to the communications bus (network) onboard the spacecraft
- Does not necessarily include ground, spacelink, or special onboard interfaces
- Scope includes all types of spacecraft, from micro and small to large and/or manned, from LEO to deep space, because a bit is a bit
- Scope also includes all types of devices from large Instruments to small engineering sensors like a temperature sensor

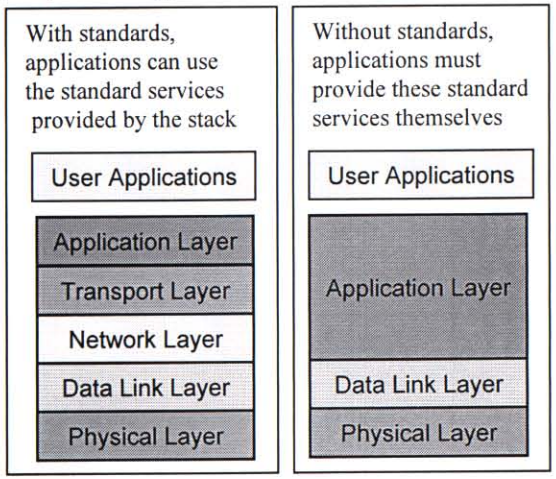


The SOIF Layered Model Is Related to the ISO Model

- User Applications use Communications Services
- Application Layer provides message transfer service and standard I/F
- Datagram service provided with internetwork routing (when needed)
- Convergence to single interface definition to hide network/interface differences from higher layers
- Network or Interface as defined by H/W designers of data bus/network

SOIF Reduces Impact on Applications

- When communications standards are used, then the application can use the services provided by the protocol
- Without communications standards, then the applications either need to supply the service themselves, or do without the service
- Without standards,when the underlying bus changes, then the effects will ripple up into the applications



SOIF Goals

- Standardized spacecraft interfaces should lead to:
  - Development of "Plug and Play" instruments, components and sensors
  - Reduced development costs and risks
  - Shorter development & integration times
  - Increased quality & reuse of flight & test equipment
  - Second-sourcing of flight and test equipment
  - Easier adoption of new and evolving technologies
    - Hardware and Software upgrades
    - Autonomy
    - Vehicle Health Management

SOIF Objectives

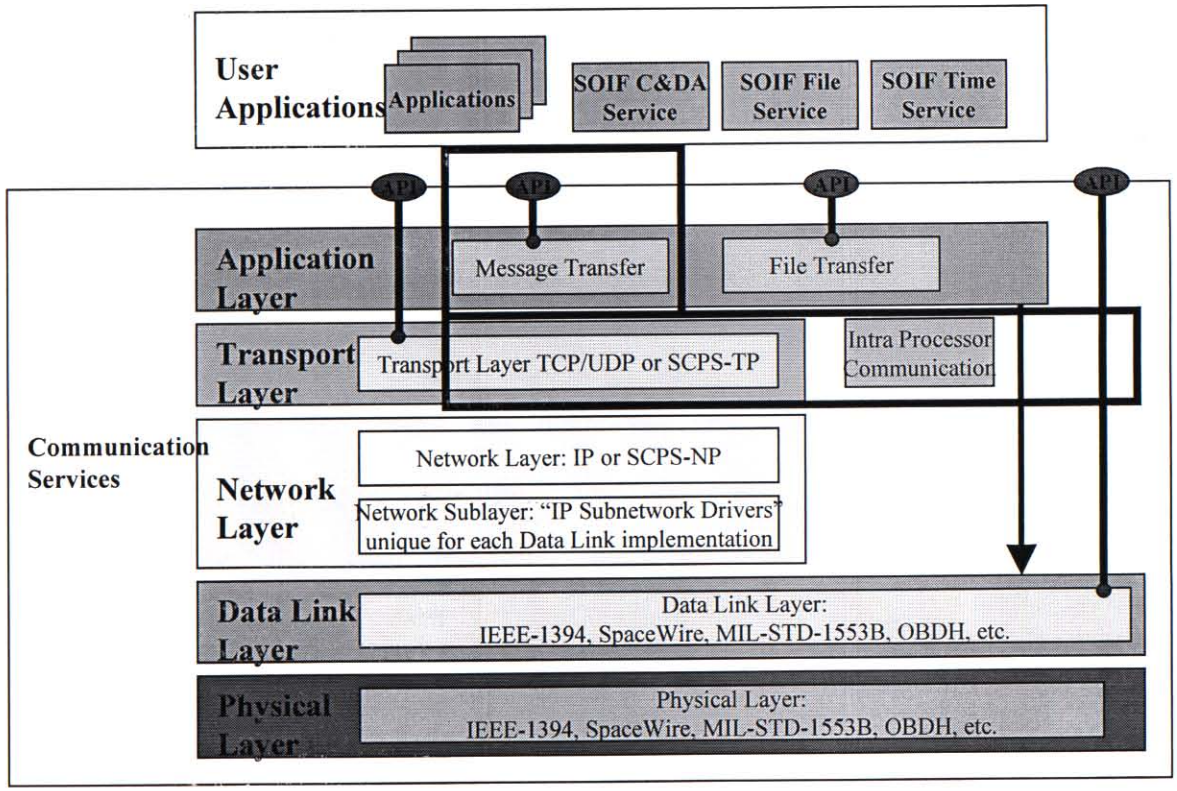
- Develop communications services for User Applications
  - That will meet the needs of spaceflight systems, without excessive overhead or resources
  - Provide standard application software I/Fs
  - Leverage modern S/C functional capabilities
- Provide a selection of protocols that make sense for spacecraft
  - TCP/IP, with selected RFCs
  - SCPS (efficient space tuned TCP/IP, where required)
  - Low overhead inter-process links as required
- Support changing the underlying data bus to meet the needs of the application
  - High Speed bus for science needs
  - Medium and/or Low Speed for command & control
  - Low Speed for sensor bus & non-critical applications
  - Can interface to wireless and long haul telecommunications links as required
- Will also support the seamless communications between elements in different nearby spacecraft, such as for constellations, formation flyers, and co-operating spacecraft

SOIF Use Cases

- **Spacecraft Environments**
  - Single Spacecraft
  - Multiple Spacecraft
    - Constellations / formation flyers
- **Bus Environments**
  - Simple single bus
  - Multiple bus, differential rates
  - Fail-over, fault tolerant
- **Bus Topologies**
  - Simple
  - Hierarchical
  - Redundant
- **Instrument Models**
  - Simple instruments, sensors, effectors
  - "Smart" instruments w/ micro-processors
- **Processors**
  - Single processors
  - Multiple processors
  - Heterogeneous processors
- **Connectivity Models**
  - Continuous (interactive)
  - *Dis-continuous (store & forward)*

Message Transfer Service Context

- Provides Message Transfer interface and confirmed & unconfirmed message delivery
  - Compliance Test Port for Message API
  - Socket I/F to top of TCP/UDP
  - Or interface to shared mem / pipes / data link



NOTE: No security protocol is shown in order to simplify this diagram

Message Transfer Service(MTS)

Definition

A set of common functions for message transfer and Quality of Service (QoS) management that sits above the transport layer in the communications stack and provides a service API for mediating the transfer of data between processes in a distributed system.

Assumptions:

- The processes run on a single system or multiple systems that may be heterogeneous
- The network can be LAN, WAN (Wire or RF) and Internet
- The Transport Layer functionality may be provided by TCP/IP or SCPS, or by a direct Data Link, or by efficient mechanisms such as shared memory, pipes, or other IPC.

High Level Requirements

The MTS specification is to provide applications with the following services:

- Transfer of information of arbitrary structure between processes over point to point or multi-point connections
- Control and specification of the quality of service and service delivery mechanisms (timeliness, reliability, throughput, determinism, reporting, policies, transfer services)
- Responsive control, monitoring and reporting of the service and/or connections
- Application portability and interoperability across heterogeneous implementations

Major Capabilities

- Name Space Lookup - to translate a process name to an address/path of some form
- Connection establishment - to link two or more processes for communication (early/late binding, service negotiation, policies)
- Synchronization - events, triggers, and asynchronous or synchronous communication
- Message Delivery - to send or receive messages
- Message Type - classes of messages that are transferred
- Transfer Mechanism - control of the transport protocol or link
- Error Handling - to detect, report, and recover from errors
- QoS - link performance and management

Preliminary Prototypes

Three independent teams developed initial prototypes to explore feasibility, functionality, performance and footprint. Only prototyped a minimum set of functionality. An initial integrated API was defined following analysis of these prototype approaches.

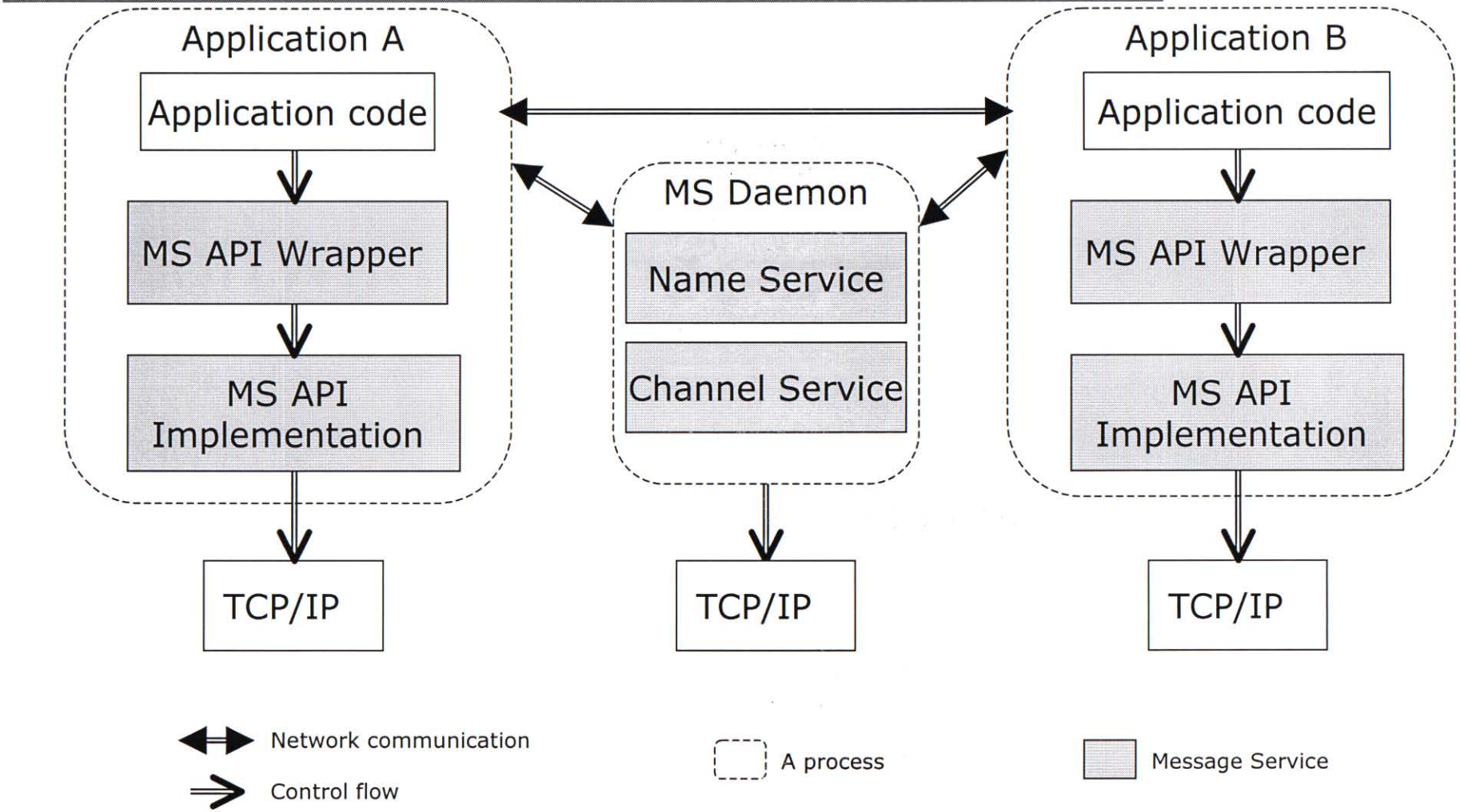
Three teams are:

- **Logica** – based on work done for a prototype SOIF Command and Data acquisition (C&DA) interface. This approach is based directly on use of UDP over a standard socket interface.

- **Science Systems Ltd** – based on work done in developing a minimal CORBA ORB for space use. Compared its General InterORB Protocol (GIOP) implementation with the agreed specifications. The GIOP was built over TCP sockets.

- **JPL** – implemented an API that borrowed elements from the Message Passing Interface Real Time (MPI/RT 1.1) specification using the Asynchronous communications Environment (ACE) toolkit. The implementation was based on TCP.

Current Prototype Software Architecture



Current API Specification*

Service Setup

initializeMsgService *(int argc[in], char *argv[in], MSHandle *msh[out])*

initialize the message service

finalizeMsgService *(MSHandle *msh[in])*

terminate the message service

resolveName *(MSHandle *msh[in], MsgEntityName entityName[in], MsgTargetAddr targetAddr[out])*

resolve the name of a message service entity to its internal address that is referenced by Message Service

bindName *(MSHandle *msh[in], MsgEntityName entityName[in])*

bind a message service entity to its name

**** Reviewed during Fall CCSDS meeting***

Current API Specification (cont)

LoS

queryLoSCapability *(MSHandle *msh[in], MsgTargetAddr *targetAddr[in], LoS *los[out])*

inquire the LoS attributes and allowed values supported by the given target address

setLoSCapability *(MSHandle *msh[in], LoS *los[in])*

set the level of service capabilities that the current process may support

Current API Specification (cont)

Communication

openChannel *(MSHandle *msh[in], MsgTargetAddr *targetAddr[in], MSChannel *channel[out], LoS *los[in], struct timeval *timeout[in])*

open a channel for communication with a peer process in a blocking mode.

closeChannel *(MSChannel *channel[in])*

close the given channel

Send/Receive (block)

sendMsg *(MSChannel *channel[in], char *buffer[in], int datalen[in], struct timeval timeOut[in])*

send a message through the channel in a blocking mode with the given message buffer and size

receiveMsg *(MSChannel *channel[in], char *buffer[in/out], int buflen[in], int *datalen[out], struct timeval timeOut[in])*

receive a message through the given channel in a blocking mode

Current API Specification (cont)

Send/Receive(nonblock)

sendMsg *(MSChannel *channel[in], char *buffer[in], int datalen[in], struct timeval timeOut[in])*

send a message through the channel in a blocking mode with the given message buffer and size

sendMsgPoll *(MSChannel *channel[in], char *buffer[in], int datalen[in], RequestID reqID[in])*

send a message without waiting for its completion. Will check the send status periodically.

sendMsgAsync *(MSChannel *channel[in], char *buffer[in], int datalen[in], RequestID reqID[in], void *handlerPtr[in])*

send a message without waiting for its completion. An event handler is optionally specified to be invoked when the send operation completes with or without an error.

receiveMsgPoll *(MSChannel *channel[in], char *buffer[in], int buflen[in], int datalen[out], RequestID reqID[in])*

receive a message through a given channel without waiting for its completion. Will check for its completion periodically

receiveMsgAsync *(MSChannel *channel[in], char *buffer[in], int buflen[in], int datalen[out], RequestID reqID[in], void *handlerPtr[in])*

receive a message through a given channel without waiting for its completion. An event handler is optionally specified to be invoked when the receive operation completes with or without an error.

Current API Specification (cont)

Event Notification (callback)

registerAsynchRequestHandler *(MSChannel *channel[in], void *handlerPtr[in])*

register an event handler specifically for asynchronous request (send/receive)

handler_func *(MSChannel *channel[in], RequestID reqID[in], char *buffer[in], int len[in])*

specify an event handling function

eventLoop *(struct timeval timeout)*

loop to receive and dispatch events, return when timeout or error

Status Test

pollChannel *(MSChannel *channel[in], RequestID reqID[in])*

check the status of a request for the given channel and requestID

Next Steps

Validation of Overall Approach

Current approach has been validated within Messaging SIG and Panel 1K.

API Specification

Review and update the the message service interface specification, including the API syntax/semantics/implication, and the control message format/protocol.

Prototype

Complete implementation of the API interface and functionality in C++ on Red Hat Linux 7.2. Perform benchmarks to verify the performance requirements.

Integration of the prototype software with MDS (Mission Data System)

Validate the feasibility of the prototype software in a realistic S/C environment

Further Enhancement

Include other interface functionalities (QoS & control, fault tolerant network, multicast) as value for applications environment is determined

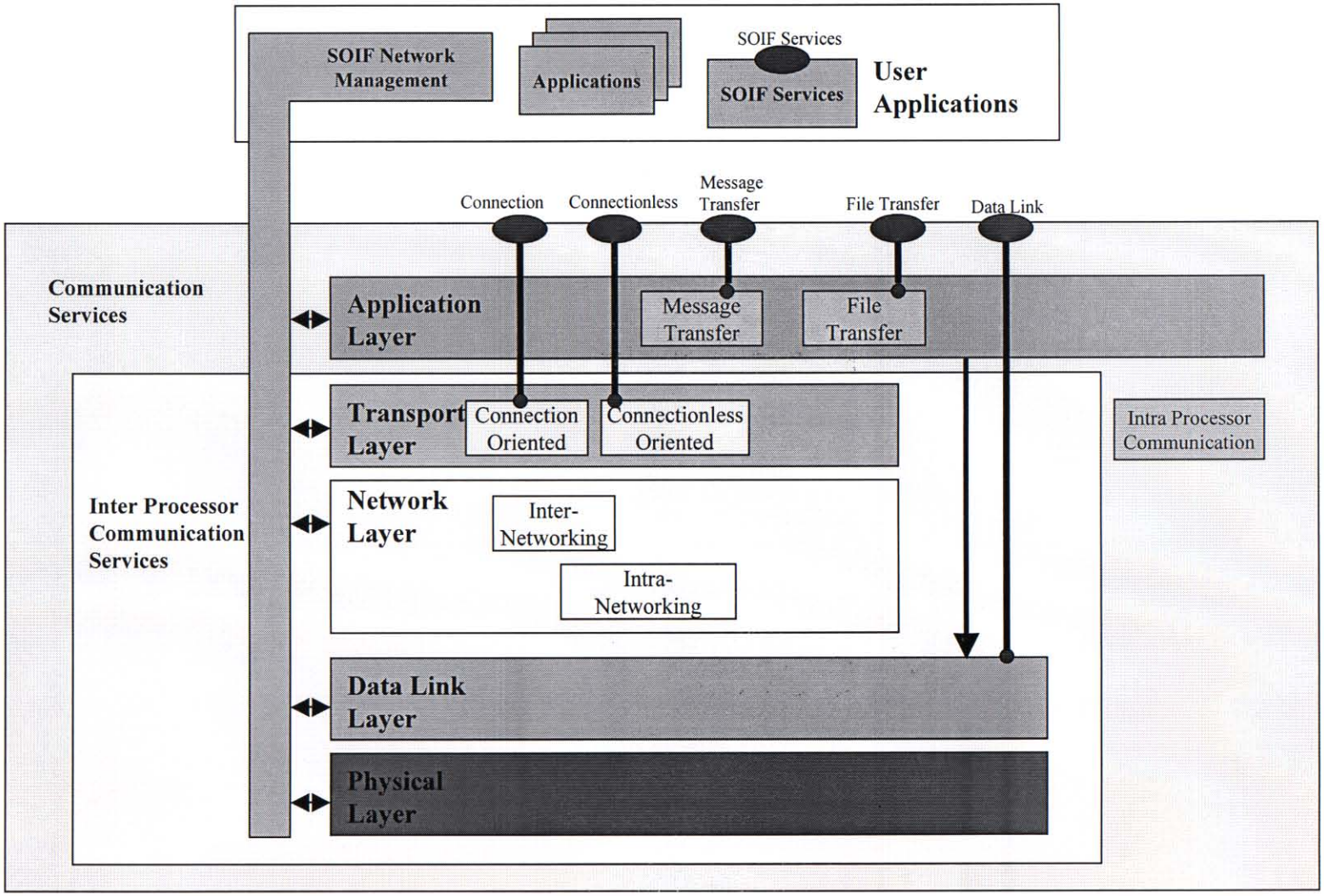
Acknowledgement

- Concepts developed within CCSDS Panel 1K, Spacecraft Onboard Interfaces (SOIF)
- Message Transfer Service developed by Messaging Special Interest Group

Peter Shames, NASA/JPL, Chair
Nick Achilleos, BNSC/Logica
Ewan Carr, BNSC/Logica
Laverne Hall, NASA/JPL
Stuart Fowell, BNSC/SciSys
Ed Greville, NASA/GSFC
Chialan Hwang, NASA/JPL
Gavin Kenny, BNSC/Logica
Norm Lamarra, NASA/JPL
Gary Lay, BNSC/Logica
Stuart Mills, ESA/U of Dundee
Nick Rouquette, NASA/JPL
Joe Smith, NASA/JPL

Backup

SOIF Reference Model



High Level Goals

1. The MTS implementation is to exhibit the following characteristics:
   - Low overhead
   - Low latency
   - Small "footprint"
   - Thread - safe
2. The MTS implementation goals include
   - Portability & reusability across several different platforms
   - Interoperability among different language and OS platforms and over different communication protocols
   - Dynamic handling of connection state and failure modes
   - Support for multi- spacecraft constellations and formations (swarms)
   - *Support for strongly typed messages & type safety checking during connection establishment (optional)*

Requirements on Lower Layers

The MTS requirements on lower layer protocols include

- Mechanisms to support early and late binding of connections
- Mechanisms to support dynamic (re-)binding of connections
- Support for resource reservation protocols such as RSVP
- Mechanisms for negotiation of connection characteristics and QoS parameters
- Means for signaling link state changes which may be link dependent (at least Red, Yellow, Green)

High-Level Functionality

- Communication Mechanisms
  - Service initiation / termination, authentication
  - Process - Process communication
    - send/receive (two-sided), put/get (one-sided), blocking/non-blocking
  - Multicast / Broadcast - specified set or promiscuous recipients
  - Publish/subscribe - managed delivery of data
  - Callback - event notification and handling
- Quality of Service
  - Service & connection negotiation, policy management
  - Priority/scheduling/resource management
  - Reliability, determinism, ordering, timeliness
- Message buffer management (make/free/get/set/init)
- Error Handling, monitoring, signaling, reporting
- Group management (joingroup/leavegroup)

- *Derived data types (optional / future)*
- *Packing/unpacking (optional / future)*

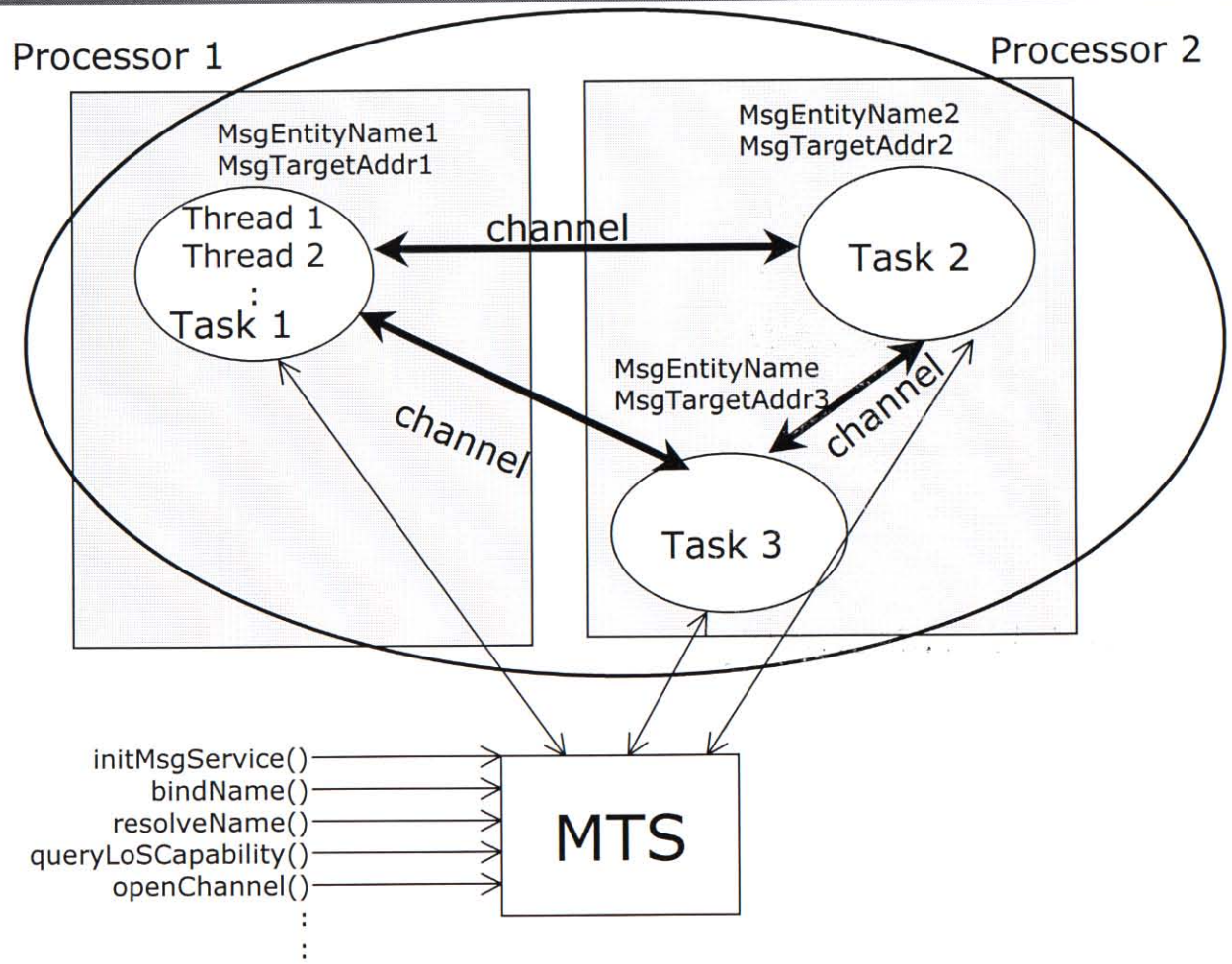
Terminology

Application	a software that runs on computer(s) to achieve a specific set of functional requirements. An application may consist of one ore more tasks that run on a single processor or multiple processors.
Channel	a logical connection between two or more applications
CoS	Class of Service, e.g. guarantee/non-guarantee, reliable/unreliable, store and forward
LoS	Level of Service is a table of CoS and QoS that are provided to channels between two message entities (peers)
MS Channel	a handle to a channel information block that is generated and uniquely assigned to a channel by MTS
Msg Entity	an application task unit that uses the Message Service to send/receive messages from another entity
Msg Entity Name	a user-friendly name to identify the entity that uses the Message Service. It can be a service function that the application provides.

Terminology (cont)

Msg Target Addr	an opaque handle for a specific message peer (entity). This handle corresponds to one and only one symbolic name.
Msg Svc Handle	a handle for the message service that is returned to a task by initializeMsgService
Name Format	the format of a name. This is hierarchical and may include the group name for the multicast purpose.
Name Service	an independent software that maps a message entity name to its target address
QoS	Quality of Service
Task	a process that owns its own address space and acts as an entity to communicate with other tasks through channels (connection)

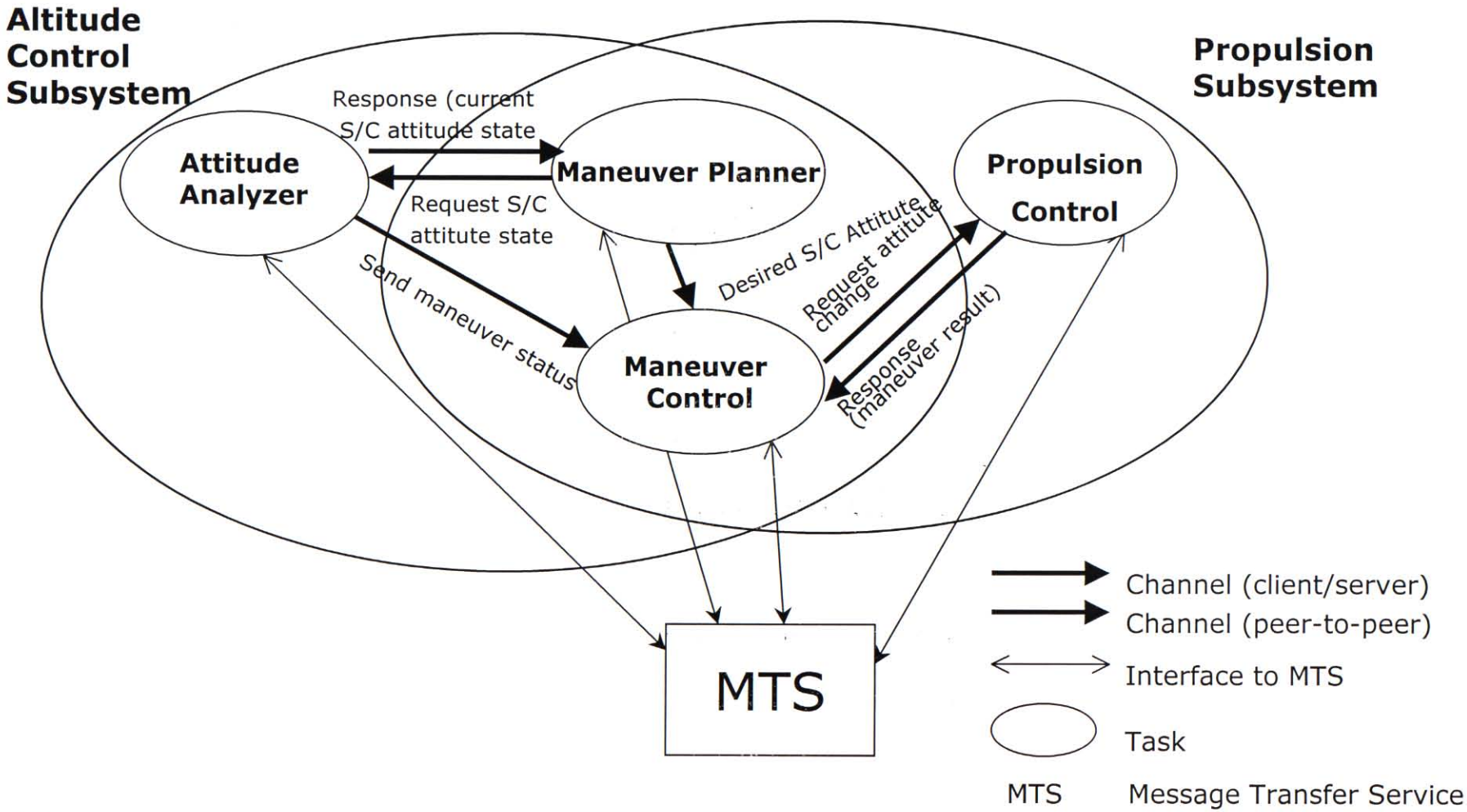
Concept Illustration



Application 1

Concept Illustration (cont)

Use Case: Client/Server and Peer-to-Peer Interaction



MTS in SOIF Context

Application Layer Interaction Model

1. Client/server (listen/connect/accept)
   C1 request / response
   C2 request / response S
2. Peer-to-peer
   P1 ⇄ P2
3. Pub/Sub
   P → S1, S2, S3

Message Transfer Service API

init()
bind()/resolve()
openChannel()
receiveMsg()/sendMsg()
:
:

Transport Layer

Shared Memory	Message Queue	PIPEUDP	TCP	CFDP
		SCPS-TP	SCPS_TP	• store & forward • Msg = File • Msg = DGRM

Data Link Layer

memory	O.S. Service	Space Link (TC & TM)	Proximity 1	Other Data Link Bus Protocol

LoS Matrix

CoS vs. QoS Matrix

CoS (when) \ QoS (how)	Reliable/unreliable	RT/NRT	Deadline/Prio/Start time
Interactive Communication (continuous connectivity) short RTLT(Round Trip Life Time)	**Reliable (ack, unack) Unreliable (unack)**	**RT**	Deadline/Prio
or			
Store-and-forward (periodical connectivity)	Reliable (ack, unack) Unreliable (unack)	NRT	Deadline/Prio/Start time
and			
Guarantee (rate, delay, bandwidth)	Reliable (ack, unack)	RT/NRT	Deadline/Prio/Start time

Message Passing API

Prototype - High-Level Functionality

- **Communication Mechanisms**
  - ***Service initiation / termination, name space lookup***
  - ***Process - Process communication (peer-peer only)***
    - ***send/receive (two-sided), blocking/non-blocking, (trigger, polling)***
  - ***Event notification and handling (Callback)***
- **Quality of Service**
  - ***Simple service & connection negotiation***
  - ***Reliable (acknowledged) & unreliable***
- **Message buffer management (make/free/get/set/init) (local issue)**
- **Error Handling, monitoring, reporting (simple)**

Prototype Evaluation

Single and dual processor configurations

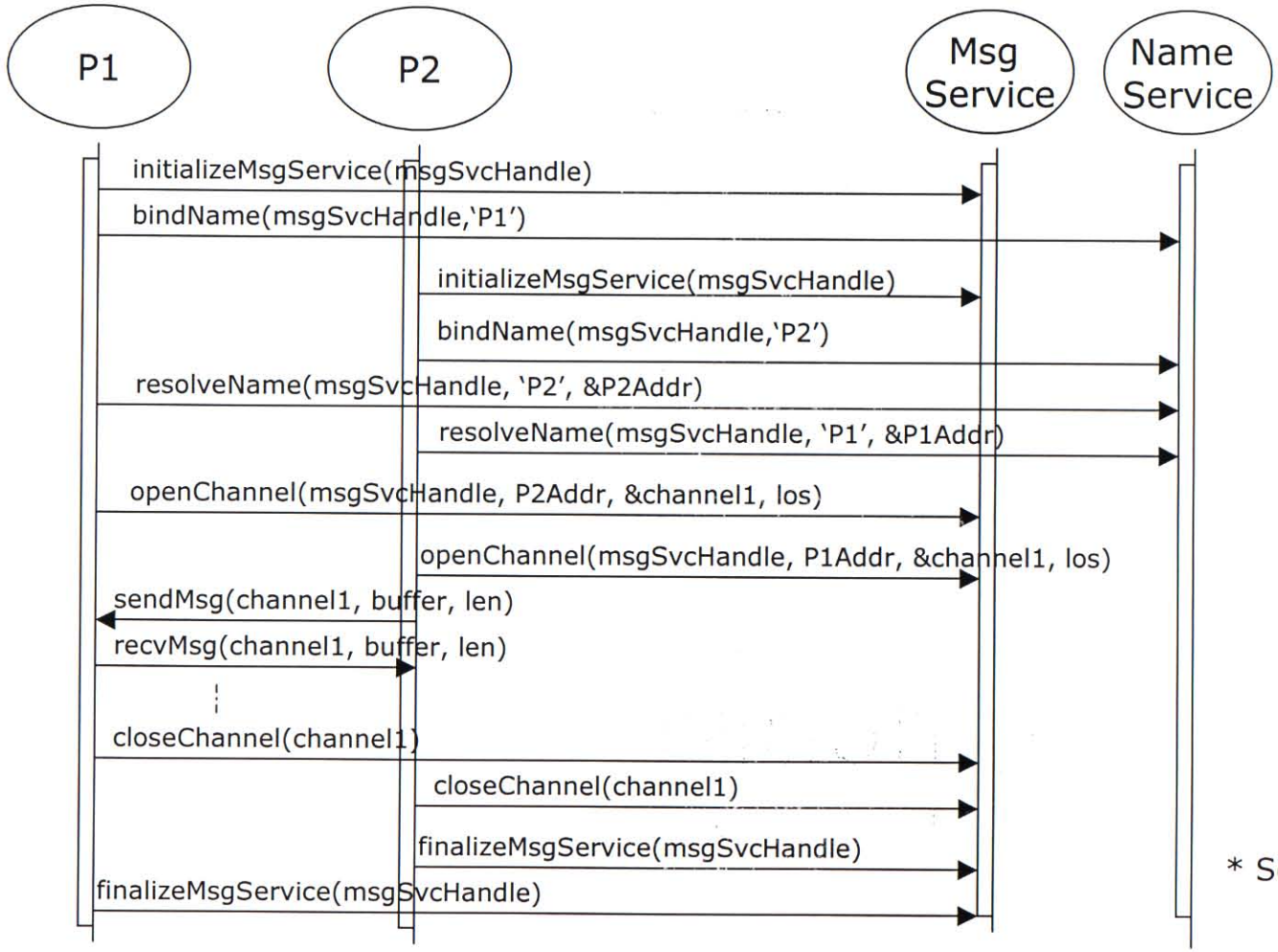
Tests emulate typical message exchanges

Measure: footprint, performance, overhead

Evaluate: functionality, ease of use, coverage of requirements

Concept Illustration (cont)

Application Level Sequence Diagram* (based on proposed API)



* See others in backup

Prototype Performance Comparisons

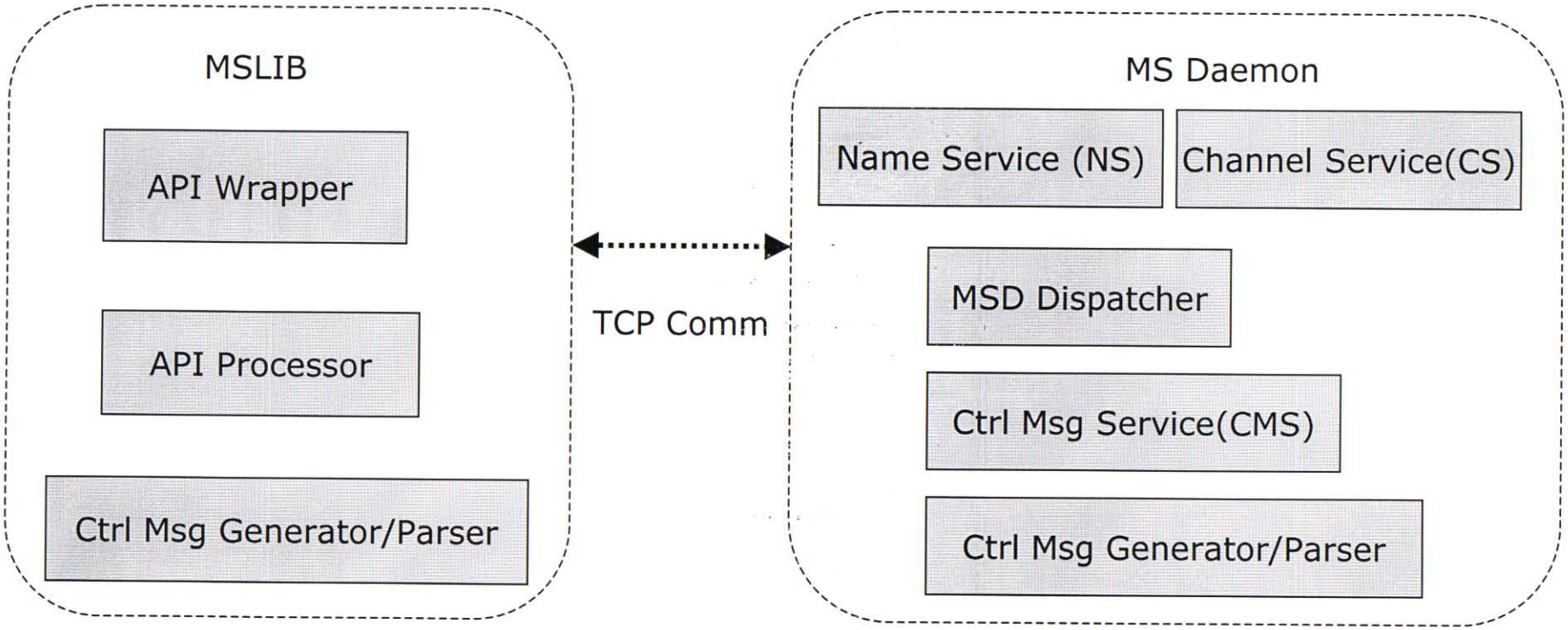
Category	SOIF Messaging Layer Functionality	JPL MPI/RT-ACE (Chialan)	SciSys MinORB/GIOP (Stuart)	Logica CDA Msg'n (Gavin/Ewan)
Performance:				
- Use Cases	Single Proc'r / Single Bus	no	yes	
	Two Proc'r / Single Bus	yes	yes	
- Message Traffic			1000 iter (100 for data xfer)	ave of 5 tries * 10K iter
Status Type	10B msg, 10B rsp	na / 1764 msg/sec	5000/1428 msg/sec	na / 2321 msg/sec
	- throughput, jitter, latency		0.2ms/0.7ms	
Control Type	10B msg, 1KB rsp	na/1564 msg/sec	3333/400 msg/sec	na / 1846 msg/sec
	- throughput, jitter, latency		0.3ms/2.5ms	
Config Type	1KB msg, 10B rsp	na / 1302 msg/sec	1000/312 msg/sec	na / 1801 msg/sec
	- throughput, jitter, latency		1.0/3.2ms	
Data Transfer Type	10B msg, 100KB rsp	na	83/7.7 msg/sec	na
	- throughput, jitter, latency		12ms/130ms	
Test max rate sync comm	msg tx count in 1 min (wait for each prior tx to complete)			
- Protocol Overhead	(message header)	n/a		
- Optimization Areas	(planned improvements)	send multiple messages at a time	use pt-pt LAN for test, interleave req/resp	
Config/Environment:	(Open Standards)			
- API Interface Type/Method		C++ objects	C func IIOP, IDL (CDA type)	c func, IDL
- Language(s)		C++	C lang, GCC 2.96	c, gcc
- Operating Sys(s)		Red Hat Linux release 7.0	RH Linux 7.1	Free BSD 4.3
- Platform(s)		Pentium III, i686	Dell Gxa P2 266, 64 MB & P2 300 64 MB	Dell Latitude P3
- Processor Speed		866MHz dual processors & 600 MHz dual processors	P2 266 MHz & P2 300 MHz	P3 800 Mhz
- Data Link / Physical Network(s)	(ex., 10 Mb vs. 100 Mb Ethernet) - affects perf.	100Mb	10 Mb	10 Mb
- Net Protocol(s)	TCP/IP (or UDP)	TCP	TCP	UDP
- Prototype Footprint	(memory size)	libsml.so = 1.136 MB		
- Support Toolset	(name/type/cost)	ACE/open source/free		
- Toolset Footprint	(memory size)	libACE.so = 8.34MB	10 KB, GIOP & IIOP	20 KB

Current Prototype Assumptions

Assumptions:

1. The message service concept is based on peer-to-peer paradigm.
2. A point-to-point "channel" is a connection between two tasks and is bi-directional.
3. There exists a Name Service (NS) in the network to handle name resolution.
4. There exists a Channel Service (CS) in the Message Service to handle channel setup. It passes information during the setup of the communication channels between two or more tasks, and maintains the information afterwards.
5. Both NS and CS run as separate entities from the application tasks. The prototype combines them into a stand-alone program named "Message Service Daemon" (MSD).
6. Other Message Service functionalities are defined in a library that is linked into the application that uses the services.

Prototype Software Elements

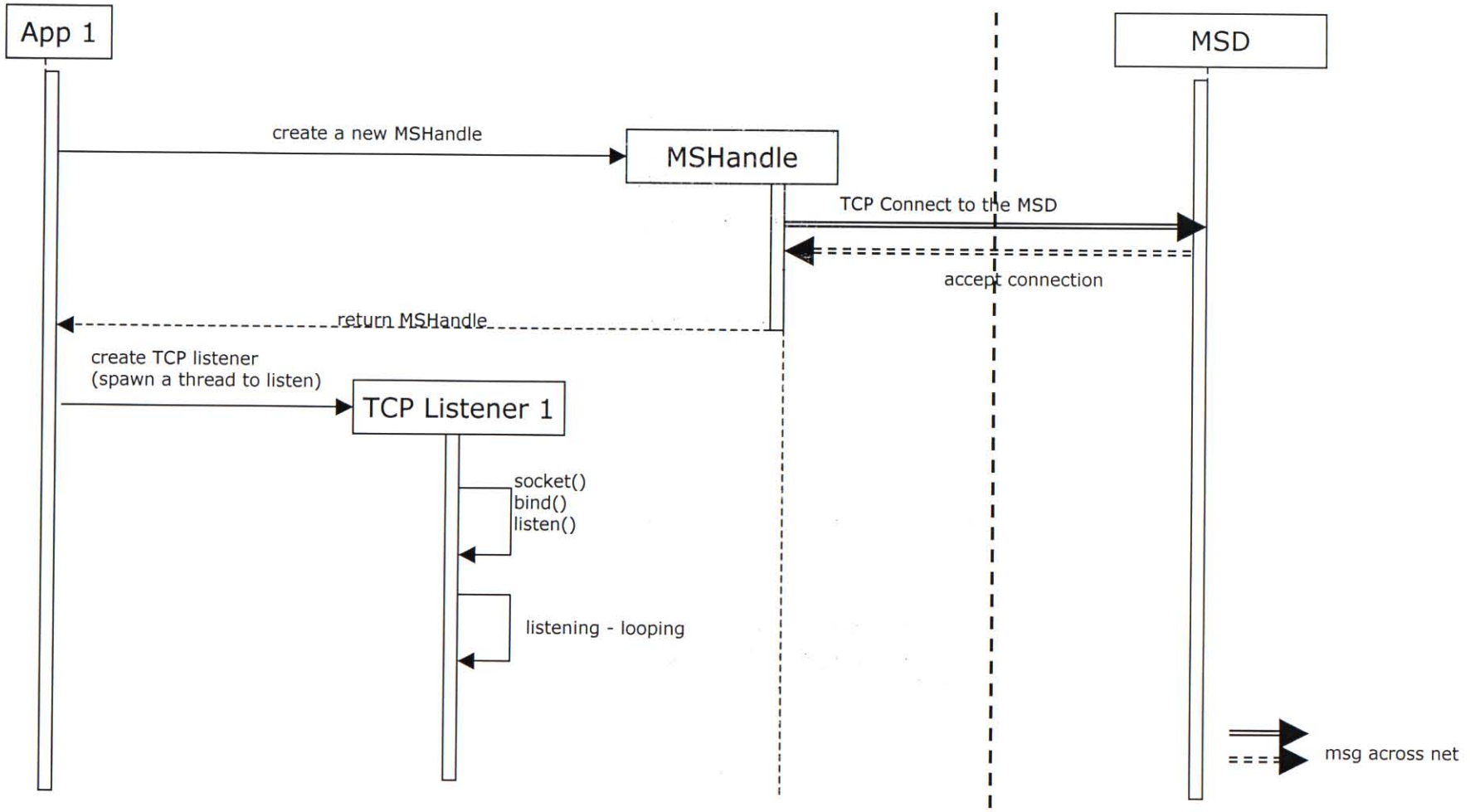


Prototype Status

1. Defined the high level software architecture and identified the major software elements.
2. Designed the sequence diagram for a initial set of API functions: initializeMsgService(), finalizeMsgService(), bindName(), resolveName(), openChannel(). See backup charts.
3. Defined the control message format and protocol for the initial set of API functions.
4. Coded for major classes and continue adding more functionality, such as:
   MSAPI, MSHandle, MSCtrlMsg, MSCtrlData, MSLoS, MSTargetAddr, MSChannel, MSTCPListener

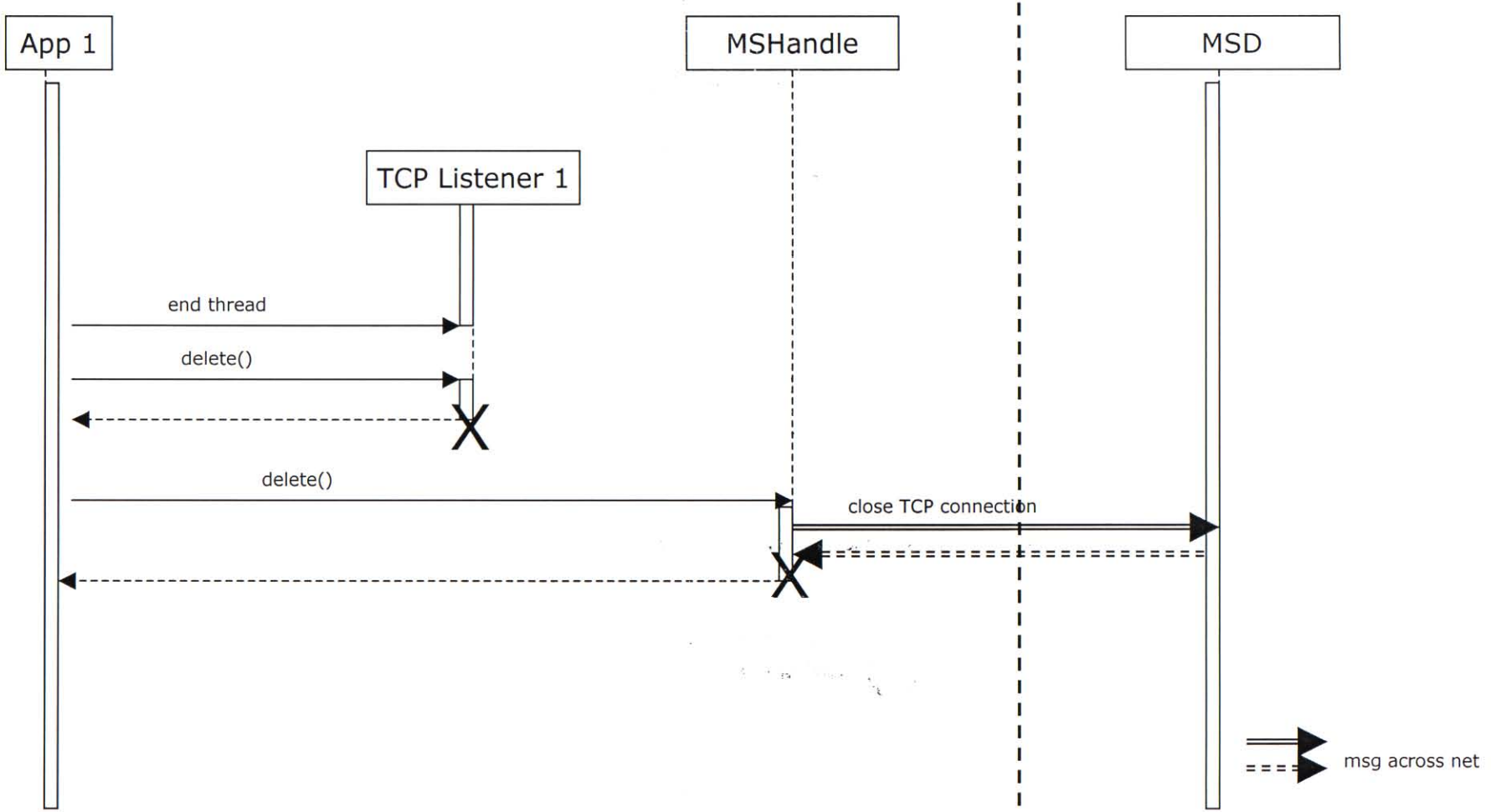
initializeMsgService()

Sequence Diagram



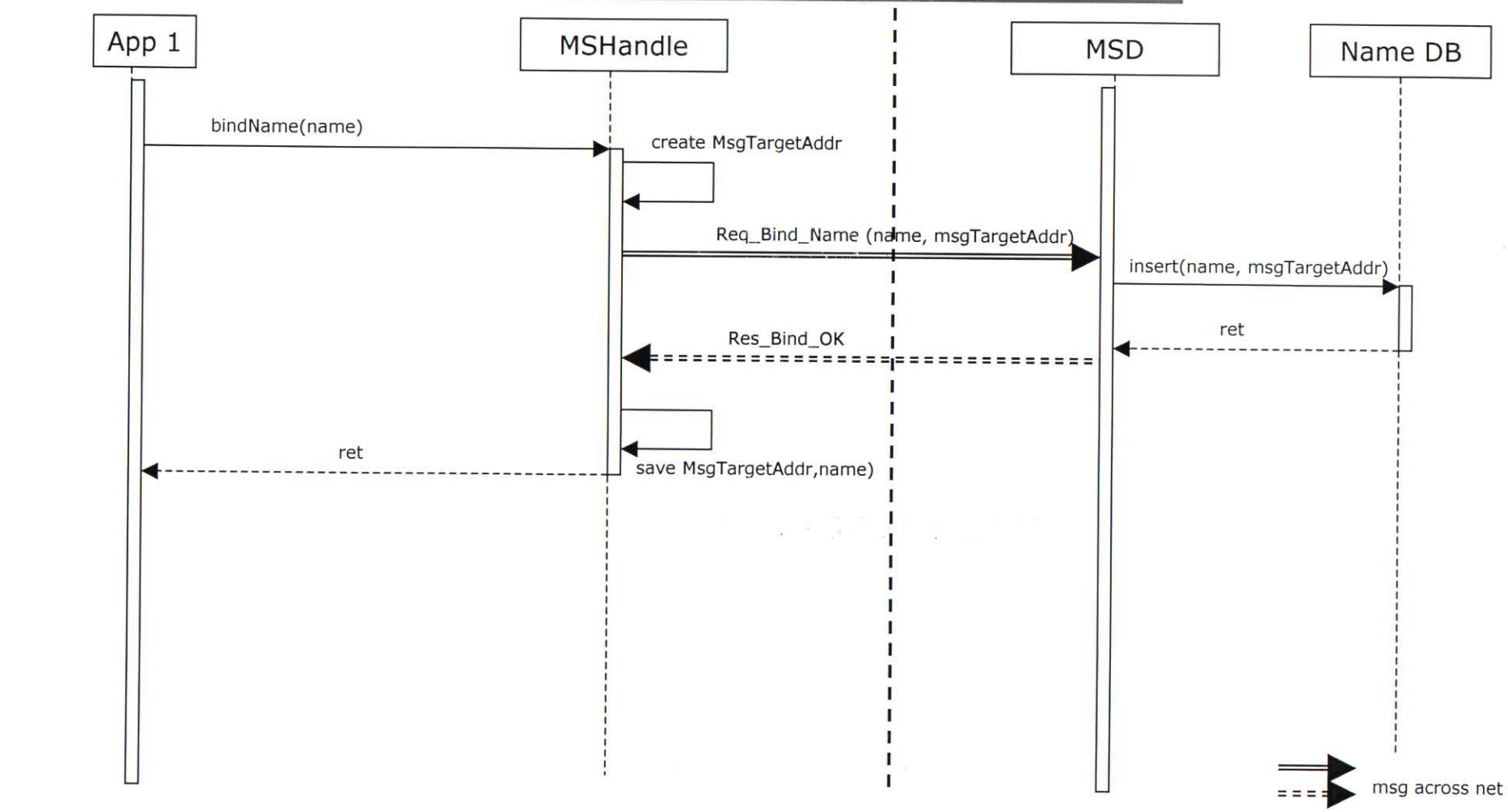
finalizeMsgService()

Sequence Diagram



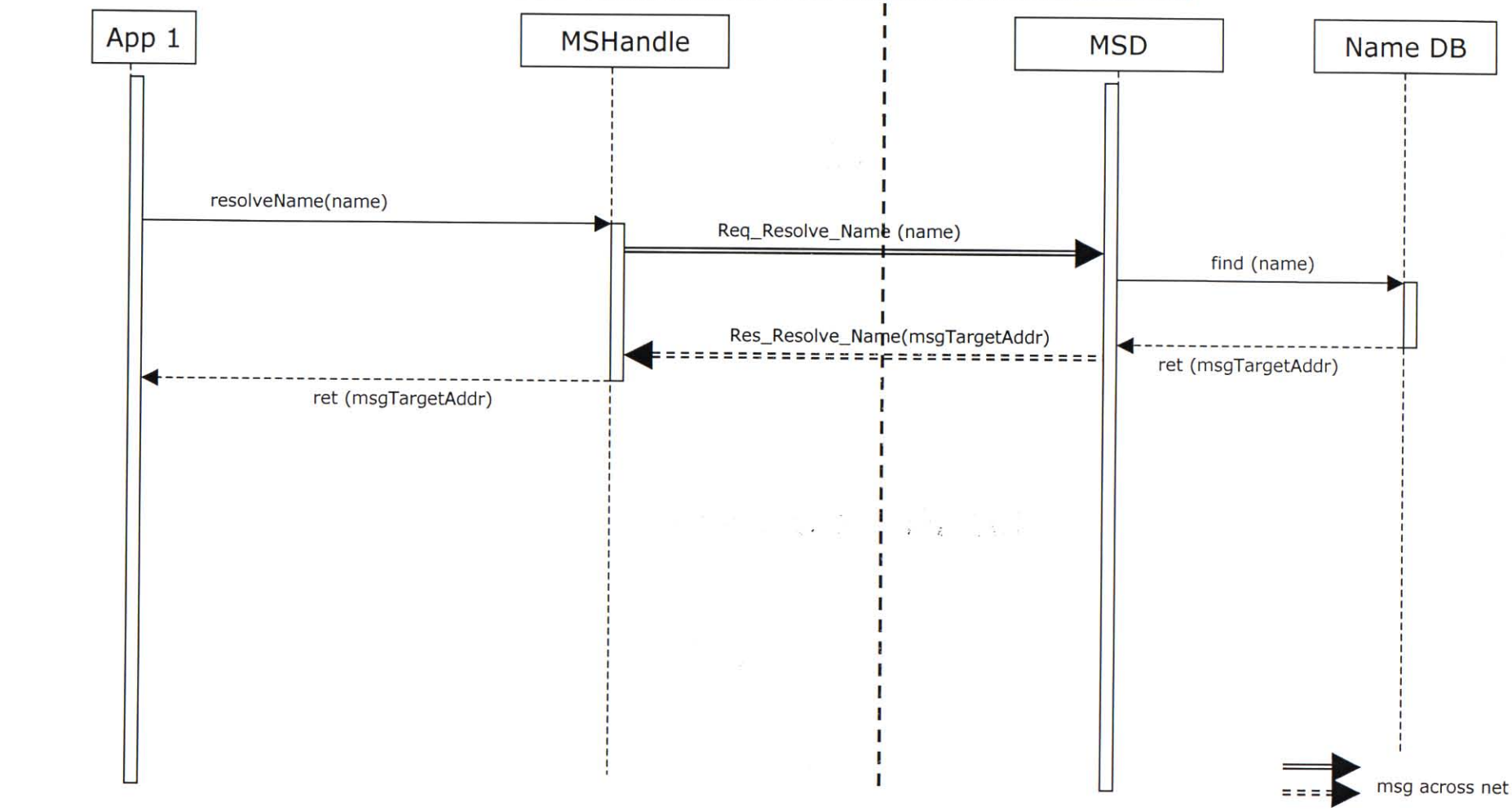
bindName()

Sequence Diagram



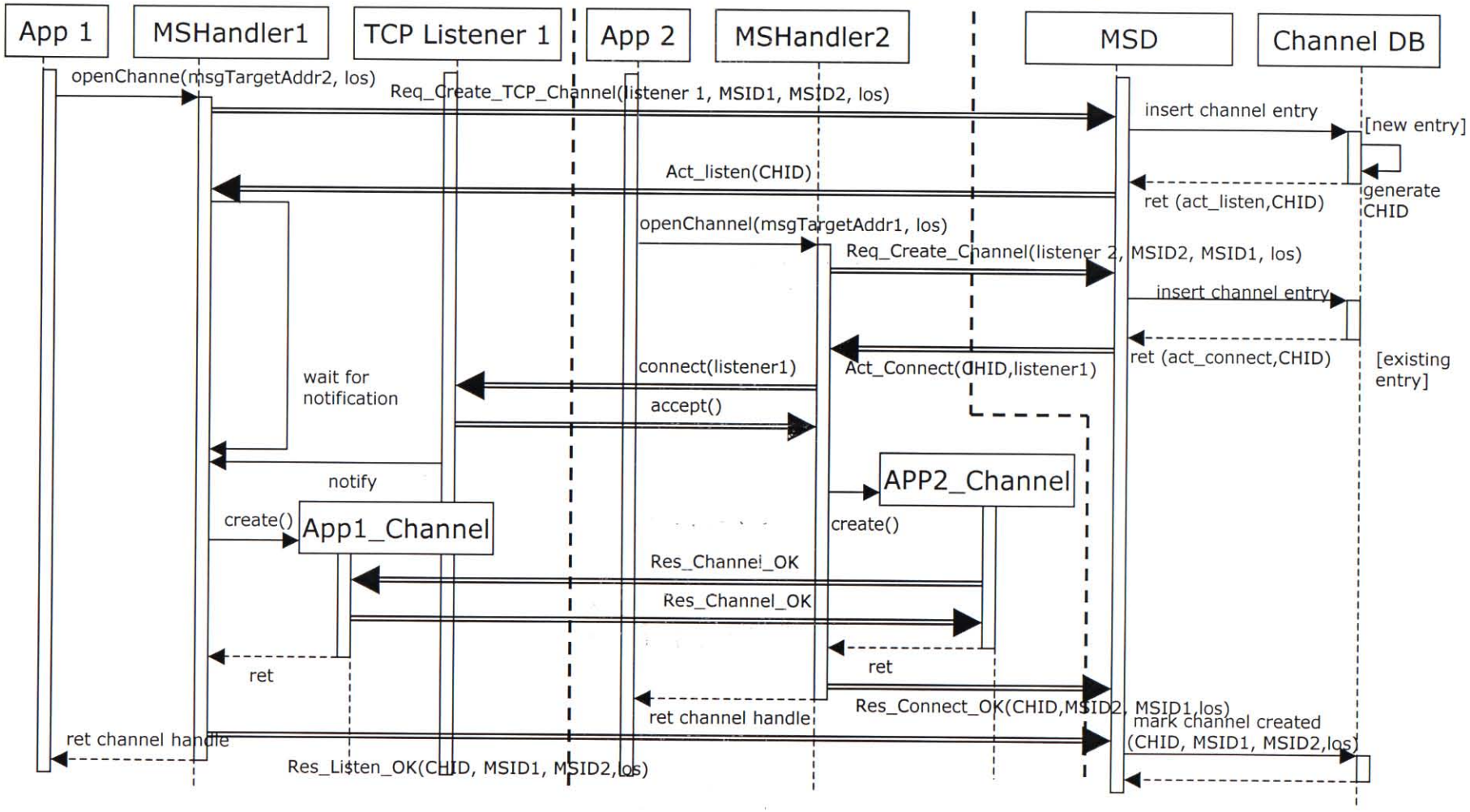
resolveName()

Sequence Diagram



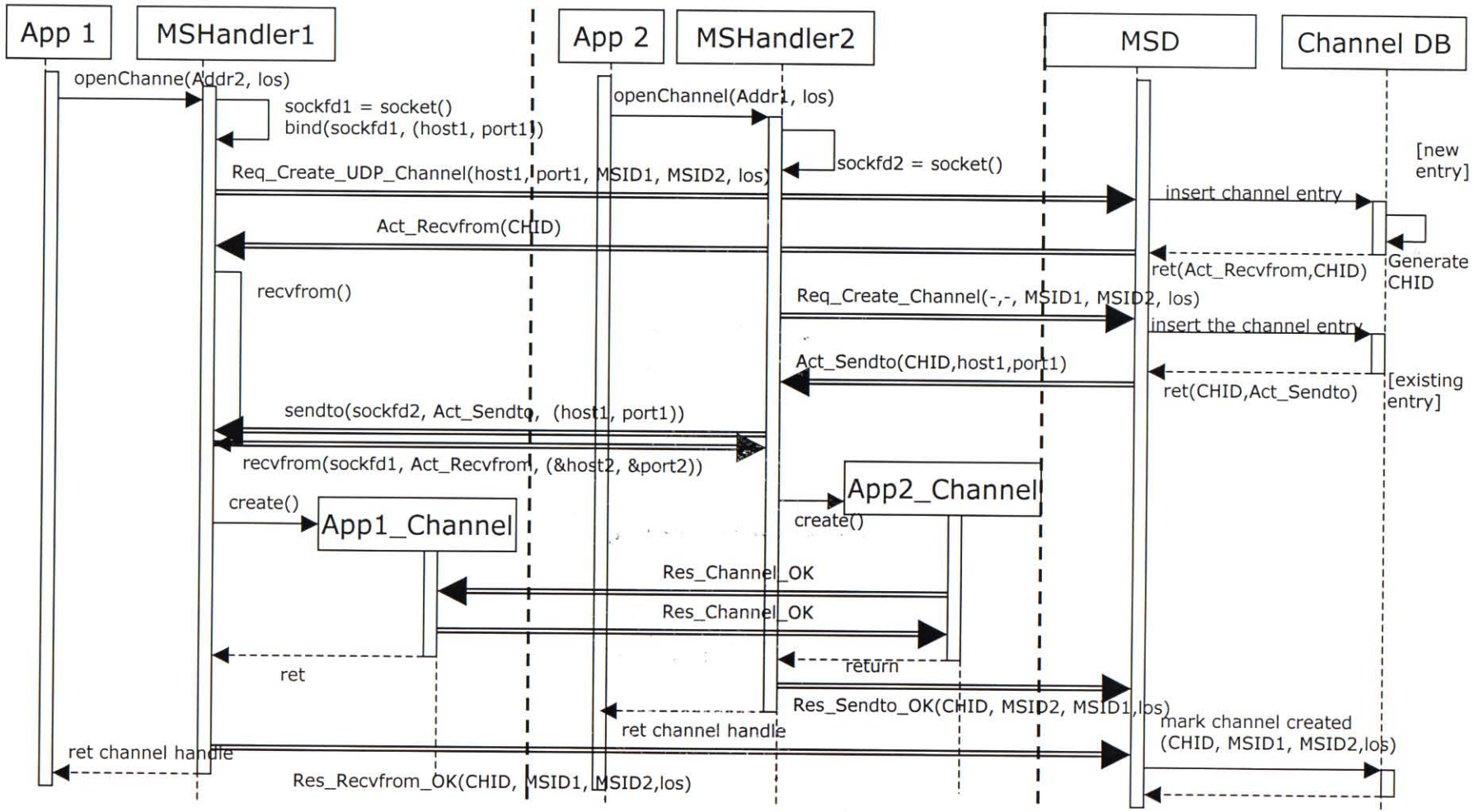
Open TCP channel

Sequence Diagram

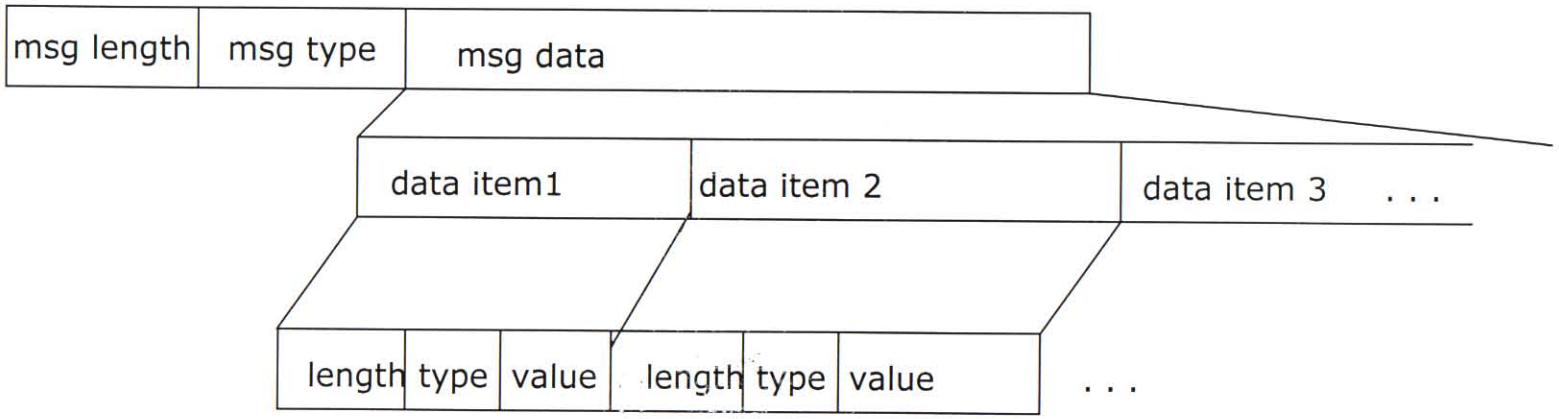


Open UDP channel

Sequence Diagram



Control Message Format



Each message has three sections:

Msg length	length of the entire message, including msg length, type and data.
Msg type	type of the message
Msg data	contents of the message. This section contains a list of data items that are designed to be part of the message type. The data items are arranged in random order.

Each data item consists of:

length	length of the entire data item (= length + type + value)
type	data type
value	data contents in ASCII format. The data format is determined by its type and should be interpreted accordingly.

Control Message

API	Msg Flow	Msg Type	Msg Data Type	Data Description
bindName	APP → MS	**Req_Bind_Name**	Data_Entity_Name	entity name
			Data_MSID	Unique entity ID assigned by Message Service
			Data_Host_IP	host IP
			Data_PID	process ID
	APP ← MS	**Res_Bind_OK**		
resolveName				
	APP → MS	**Req_Resolve_Name**	Data_Entity_Name	entity name
	APP ← MS	**Res_Resolve_Name**	Data_MSID	Msg Service ID
			Data_Host_IP	host IP
			Data_PID	process ID
			Data_Entity_Name	entity name

Control Message (cont)

API	Msg Flow	Msg Type	Msg Data Type	Data Description
Open TCP Channel	APP → MS	**Req_Create_TCP_Channel**		
			Data_Host_IP	my host IP address
			Data_TCP_Listener	my TCP listener port
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel
	APP ← MS	**Act_TCP_Listen**	Data_CHID	unique channel ID assigned by Message Service
	APP ← MS	**Act_TCP_Connect**	Data_CHID	unique channel ID assigned by Message Service
			Data_Host_IP	host to connect
			Data_TCP_Listener	listener port to connect to
	APP → APP	**Res_Channel_OK**		
	APP → MS	**Res_TCP_Listen_OK**	Data_CHID	unique channel ID
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel
	APP → MS	**Res_TCP_Connect_OK**		
			Data_CHID	unique channel ID
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel

Control Message (cont)

API	Msg Flow	Msg Type	Msg Data	
Open UDP Channel	APP → MS	**Req_Create_UDP_Channel**		
			Data_Host_IP	my host IP address
			Data_UDP_Port	my TCP listener port
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel
	APP ← MS	**Act_Sendto**	Data_CHID	unique channel ID
			Data_Host_IP	host to send data
			Data_Port	UDP port to send data
	APP ← MS	**Act_Recvfrom**	Data_CHID	unique channel ID
	APP → MS	**Res_UDP_Sendto_OK**		
			Data_CHID	unique channel ID
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel
	APP → MS	**Res_UDP_Recvfrom_OK**		
			Data_CHID	unique channel ID
			Data_MSID_Src	source
			Data_MSID_Dest	destination
			Data_LoS	LoS for the channel

Terminology

General

Broadcast Send messages to all destinations, mostly in a subnet, simultaneously

Multicast Send messages to well-defined groups of hosts that are numerically large in size but small compare to the network as a whole. To do multicasting, group management is required, including create/destroy/join/leave.

Point-to-point Send message to a destination through network connection

Group

A set of hosts that form a group for multicasting. It is possible for members of the group to be spread across separate physical networks. The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

Terminology

General

TCP (Transmission Control Protocol)
Connection-oriented transport protocol in the Internet protocol suite. It provides a reliable end-to-end byte stream transfer over an unreliable internetwork.

UDP (User Data Protocol)
Connectionless transport protocol in the Internet protocol suite. A UDP header contains the source port, destination port, UDP length and UDP checksum.

Port The TCP name for a Transport Service Access Point (TSAP.) It is a 16-bit number local to the host. Combining the port number and the IP address of the host uniquely identifies an end point for one end of the TCP connection.

Store-and-forward

A complete message is received before it is passed on to the next node. This means that each message is using at most one interprocessor link at any time but may require more storage buffers on intermediate nodes than the alternative.

Terminology

General

Reliable Communication

Communication where messages are guaranteed to reach their destination **once only**, with the **complete,** and **uncorrupted** data **in the order** they were sent. This reliability can be built on top of an unreliable protocol by adding sequencing information and some kind of checksum or cyclic redundancy check to each message or packet. If the communication fails, the sender will be notified. Transmission Control Protocol (TCP) is a reliable protocol.

Unreliable Communication

Communication where messages are not guaranteed to reach their destination reliably. The messages may not reach its destination or be received **more than once**, received **partially**, with data **corrupted**, or **not in order**. User Data Protocol (UDP) is a unreliable protocol.

Guaranteed Delivery

Communication delivery with guaranteed delay and bandwidth (or guaranteed start time, priority, and or deadline), firm (mathematically provable) bounds on end-to-end datagram queueing delays and N.B. Max delay bounds must include transmission path delays

Reference RFC 0793, RFC 2357, RFC 2212

Terminology

General

Client/Server Model

A pattern of interaction among cooperating applications. In this model, client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server. Client/server is just one approach to distributed computing.

Client

A task that sends a request to a server and waits for a response

Server

A task that receives service requests that arrive over the network, perform the service and return the result to the requester. A server can be either an iterative server or a concurrent server.

Peer-to-peer Model

A pattern of interaction among cooperating applications as equal entities. This implies a symmetric relationship between two peers in establishing connectivity and exchanging messages.

TCP-UDP Comparison

Socket	TCP Socket	UDP Socket, unconnected
connection	Connection-oriented	Connectionless
Connection Establishment	3-way handshake (listen/connect/accept)	
Reliability	Reliable ■ once only ■ complete ■ uncorrupted ■ in order	Unreliable ■ may be received more than once ■ may be missing data ■ may be loosing ■ may be out of order
Send	send()	sendto(sockfd, buff,nbytes,flags,*to,addrlen)
Receive	Recv()	Recvfrom(sockfd,*buff,nbytes,flags,*from,*addrlen)
Data transmission	Byte stream	datagram
Message boundary	No	Yes
Data checksum	Yes	Optional
Positive ack.	Yes	No
Timeout and rexmit	Yes	No
Duplicate detection	Yes	No
Sequencing	Yes	No
Flow control	Yes	no

RFC 0793 - Transmission Control Protocol

- TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols.
- TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.
- As long as the TCPs continue to function properly and the internet system does not become completely partitioned, no transmission errors will affect the correct delivery of data.

RFC 2357 - Evaluating Reliable Multicast

- "Reliable multicast transports add delivery guarantees, not necessarily like those of reliable unicast TCP, to the group-delivery model of multicast"
- "the meaning of reliability varies in the context of different applications."
- "some applications require that message delivery obey a total ordering while others do not.
- "Some applications have many or all the members sending data while others have only one data source"

RFC 2212 - Guaranteed Quality of Service

- Guaranteed service (guaranteed delay and bandwidth) in the Internet. Guaranteed service provides firm (mathematically provable) bounds on end-to-end datagram queueing delays.
- Guaranteed service does not control the minimal or average delay of datagrams, merely the maximal queueing delay. Furthermore, to compute the maximum delay a datagram will experience, the latency of the path MUST be determined and added to the guaranteed queueing delay.

Questions from RFC 2357 - Evaluating Reliable Multicast

The following questions should be answered:

- How scalable is the protocol to the number of senders or receivers in a group, the number of groups, and wide dispersion of group members?
- Identify the mechanisms which limit scalability and estimate those limits.
- How does the protocol protect the Internet (IPN) from congestion? How well does it perform? When does it fail?
- Under what circumstances will the protocol fail to perform the functions needed by the applications it serves?
- Is there a congestion (flow, QoS) control mechanism? How well does it perform? When does it fail?
- Note that congestion (flow) control mechanisms that operate on the network more aggressively than TCP will face a great burden of proof that they don't threaten network stability.

Prototype Comparisons

Category	SOIF Messaging Layer Functionality	JPL MPI/RT-ACE (Chialan)	SciSys MicroORB/GIOP (Stuart)	Logica CDA Msg'n (Gavin/Ewan)
Major Capabilities	name space	X	X	
	connection	p-to-p, broadcast	invocation (sync, callback, poll)	p-p
	synchronization	synch/asynch	tcp blocking	nb w/ polling
	message delivery	X	X	X
	message type		X	byte stream
	protocol	TCP, UDP	TCP now, POSIX msg queues	UDP
	error handling	X	X	X, status
	QoS		X	spec on open chan (not impl)
Notes			ORB w/ direct invocation intra-proc (future)	
Service Establishment	service initiation/termination	MPIRT_INIT(argc,argv), MPIRT_FINALIZE()	GIOP__Init(argc,argv), GIOP__fini	attachService(serviceID, associatioID), detachService(serviceID, associatioID), multiple associations, early bind
	Name Space Lookup	app names to opaque addr	client: Resolve_Name(*name, list of addr profiles), server: Bind_Name(), static or dynamic tables	static table map entities (log addr), to IP addr
			NB, must deal w/ net canonical form vs reciever makes right	
Communication Mechanisms	connection establish	MPIRT::Ptchannel::Create(chanID, Direction), MPIRT::Bcast_Channel::Create (chanID, Direction), Activate(), Deactivae()	Open_Connection(connection, list of addr profiles), Close_Connection()	openChannel(associationID, logicaladdress, Qos), closeChannel(associationID, logicaladdress) (multiple pairs of assoc&log addr), QoS (nil), also getLogicalAddress(applicatioID, taskID)

Prototype Comparisons

Category	SOIF Messaging Layer Functionality	JPL MPI/RT-ACE (Chialan)	SciSys MicroORB/GIOP (Stuart)	Logica CDA Msg'n (Gavin/Ewan)
	service authentication	ACE_SSL_SOCK_*		
	send/receive (block)	Insert(messageBuffer), MPIRT_START() on all channel endpoints, wait(waitflag), B/NB	Sender synch: Send_Message(connection, *buffer, len), Receive_Replymessage(conn, *requestID, *status, **buffer, *len, *swap), Reciever: Accept_Requestmessage(*con n, *replyexpected, **buffer, *len, *swap)	
	send/receive (nonblock)		Sender asynch: requestID=Send_Message_As ynch(connection, *buffer, len, *handler_func), others as in synch	sendData(associationID,logical address, data, len), readData(associationID,logical address, status, data, len) NB use polling loops using peek at data
	put/get(block/nonblo ck)	MPIRT_START() on sender, MPIRT_ACTIVATE() on receiver, B/NB	put is one-sided I/o, no get	
	multicast	MPIRT::Bcast_Channel		
	broadcast	MPIRT::Bcast_Channel		
	publish/subscribe			

Prototype Comparisons

Category	SOIF Messaging Layer Functionality	JPL MPI/RT-ACE (Chialan)	SciSys MicroORB/GIOP (Stuart)	Logica CDA Msg'n (Gavin/Ewan)
	event notification(callback)	ACE_Reactor::notify(), ACE_Reactor::handle_event(), ACE_Event_Handler, ACE_Reactor::register_handler()	send asynch sets reply handler, handler_func(conn, requestID, status, *buffer, len, swap), server: Register_Asynch_request_handler(*handler_func), handler_func(conn, requestID, reply_expected, *buffer, len, swap), Event_Loop(timeout,	monitorDataQueue (not impl)
	status test	MPIRT::<channel>::Test()	Pending_Work(), Perform_Work()	peekAtData (associationID, logicaladdress, len, nobytes) (used as polling func)
Quality of Service	service negotiation	MPIRT_COMMIT()	Check CORBA RT-ORB / GIOP spec	
	connection negotiation	MPIRT_COMMIT()		
	policy	MPIRT_BUFITER_LIFO, MPIRT_BUFITER_UNORDERED, etc.		
	priority	QoS_channel::Prtiev		
	scheduling	X		
	resource	MPIRT_COMMIT()		
	reliability	X		

Prototype Comparisons

Category	SOIF Messaging Layer Functionality	JPL MPI/RT-ACE (Chialan)	SciSys MicroORB/GIOP (Stuart)	Logica CDA Msg'n (Gavin/Ewan)
	determinism	MPIRT_COMMIT()		
	ordering	MPIRT_BUFITER_FIFO, etc.		
	timeliness	MPIRT::QoS::Time		
Message Buffer Management	**make/free/get/set/init**	MPIRT::Buffer, MPIRT::Bufiter	app supplies buffer allocated on stack, no pool, GIOP adds hdr info, Generate_Request_Msg(conn, response_expected, *RequestID, **buffer, *len), Generate_Reply_Msg(conn, status, *RequestID, **buffer, *len)	Msg Transfer internally manages buffer pool, copy user msgs into queue
Derived Data Type		X		
Packing/Unpacking			read/write data types into user supplied buffer	
Group Management	join/leave	MPIRT::Container::Cvector_Base::Group, MPIRT_GROUP_WORLD		createGroup(), addToGroup(), remFromGroup() group (not impl)
Utility	**error handling**	MPIRT_SUCCESS, MPIRT_ERR_INV_ARGUMENT	env var struct w/ msg svc errors on all func calls, few other errors params	minimal error handling
	error reporting	MPIRT_ERROR_STRING()	exception conditions flagged in env var struct	comprehensive error reporting
	error monitoring	MPIRT_PROBE		