Real-Time Multicasting for Group Communication in the Spring Kernel

Spring Project Documentation

Marie Sie-Een Teo

July 1995

1 Introduction and Motivation

Within a distributed system, process groups are a natural abstraction for performing related computations. For example, a group of server processes providing a service to clients would be a process group. To support process groups, a group communication mechanism is required to allow a process to send messages to all processes in a group, and for all processes in that group to receive that message. The most efficient way to do group communication is multicasting, where only the intended recipients are potential receivers of a message. Broadcasting requires all processes to receive and then check if the message is meant for the process, and unicasting consists of multiple 1-to-1 communications. Although we can implement group communication using broadcasting and unicasting, they are clumsy and inefficient and multicasting is preferred.

Traditional distributed systems have many differing designs and semantics for process groups and group communication [10]. The general abstraction of process groups is as a group of processes performing a related task. Current real-time operating systems which support process groups are HARTOS [3] and Spring [9].

HARTOS is a fault-tolerant real-time operating system built on an architecture of multiprocessor nodes connected in a hexagonal mesh. Processes are replicated to provide fault-tolerance and replicated processes are placed in process groups. There is some form of multicasting, messages can be sent to a distribution list associated with a process group and each member process will then be sent a copy.

The Spring real-time kernel is designed as the basis for a flexible, predictable and distributed real-time operating system. One of the goals of Spring is to provide end-to-end scheduling for processes as well as groups of processes. Many real-time applications can be decomposed into groups of processes with precedence constraints and an overall deadline (e.g. placing a component involves lowering the robot arm and releasing the component). The Spring compiler and scheduler supports the process group abstraction allowing the programmer to define precedence constraints and timing deadlines over groups of processes and nested groups.

Spring supports the general abstraction of process groups, HARTOS process groups do not support groups of different processes and hence they are a more limited notion of process groups. As seen in Spring, support for process groups provides another level of granularity, allowing coarse concurrency (if all processes in a group are active at the same time) and added flexibility (if constraints are imposed on the group and not on individual processes). That as well as the fact that real-time operating systems are becoming distributed, creates the need for us to consider a real-time group communication mechanism.

Currently, for synchronous communication we provide only $1-to-1$ semantics. Adding group communications will provide $1-to-N$ semantics. Currently, for asynchronous communication, we provide $1-to-1$, and $N-to-1$ messaging capability. Group multicasting will add asynchronous $1-to-N$ capability to Spring.

As with any form of real-time communication, the key concern for real-time multicasting will be to satisfy overall deadlines for the processes involved. End-to-end communication delays for multicasts must then be predictable in order to provide and verify timing correctness.

We begin by discussing the semantics required for a real-time multicast mechanism, discussing each issue with respect to Spring. Finally, we present the Spring design and implementation of a real-time multicasting facility.

2 Existing Communication Semantics in Spring

IPC primitives provided by Spring currently are for:

- asynchronous send and receive, where the sender does not wait for the receiver nor does it wait for the message to be queued and the receiver does not wait for messages if none are available
- synchronous send and receive, where the sending process suspends until after the receiving process does a receive on the message

Asynchronous messages are delivered by best effort, and synchronous message are guaranteed. To provide bounded synchronous communication, the system decomposes processes into tasks, and schedules tasks so that the synchronous communication is guaranteed within a bounded time.

Figure 1 shows the mapping. This is the simplified view, please see Section 5 for details. Processes A and B are synchronously communicating. The Spring compiler decomposes A and B into the tasks with the indicated (see arrows) precedence constraints. **CD** indicates that there is a time delay of at least the worst case propagation time to send a message.

Figure 2 shows the mapping for process A and B asynchronously communicating. This time there are no precedence constraints formed. However, a best-effort is made to ensure that A and B meet their deadline and the message is received by B.

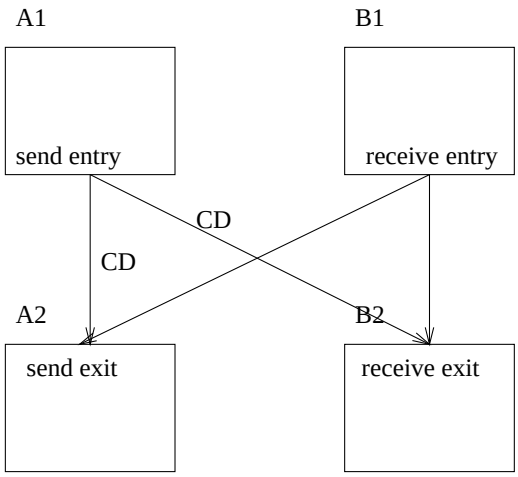


Figure 1: Synchronous Communication

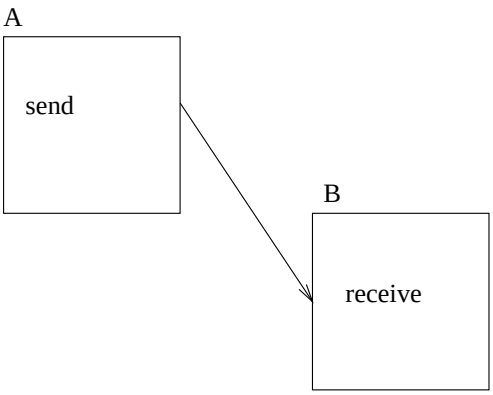


Figure 2: Asynchronous Communication

3 Group Communication Semantics in Spring

A *process group* in Spring is a collection of processes that works towards a common goal. Processes in a process group may or may not communicate. Hence, processes that will communicate are in a process group. The System Description Language (SDL) provides the language support for process groups, allowing the programmer to predefine groups and impose precedence and timing constraints on the groups.

Depending on if the communication is synchronous or asynchronous (non-blocking asynchronous), there are two types of multicasting process groups having different attributes. Figure 3 shows this.

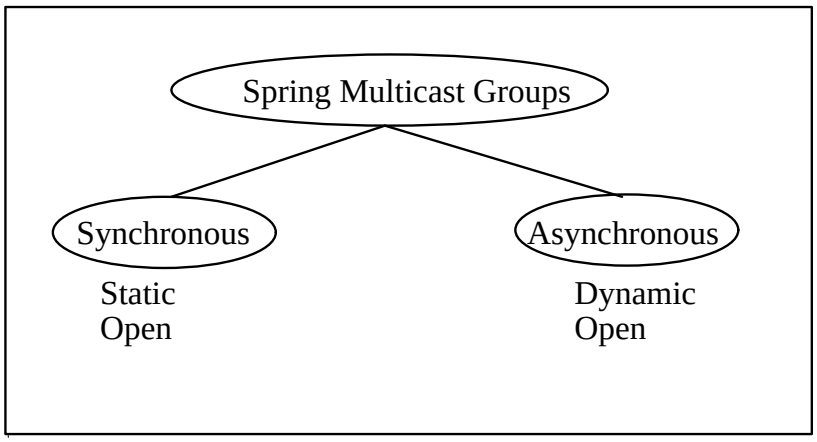


Figure 3: Definition of Multicast Groups in Spring

The attributes state whether the communication groups are:

- dynamic vs. static:
  Dynamic groups have variable membership, processes can enter and leave a group dynamically. **Static** groups have fixed membership. Here, some confusion may arise because of the fact that all process groups are predefined, using SDL. This means all multicast groups are predefined as process groups. This is not an issue for static groups. However, with dynamic groups, we must treat membership in a multicast group as separate from membership in a process group. The process group is merely the maximum set of potential multicast group members, which can then dynamically join the multicast group.

- open vs. closed:
  In **closed** groups, only the members of a group can multicast to the group. In **open** groups, any process can multicast to a group.

3.1 Synchronous Multicast Groups

As explained previously, $1 - to - 1$ synchronous communication in Spring does not actually cause the processes involved to block, the scheduler just ensures that the tasks performing

the communication are scheduled at the appropriates times so that a message is always ready when the receive is performed and the semantics are preserved. $1 - to - N$ semantics for synchronous multicasts would be provided in a similar manner. For example, if A multicasts to B and C, B and C should be scheduled such that they execute the receive only when the message has arrived, as shown in Figure 4.

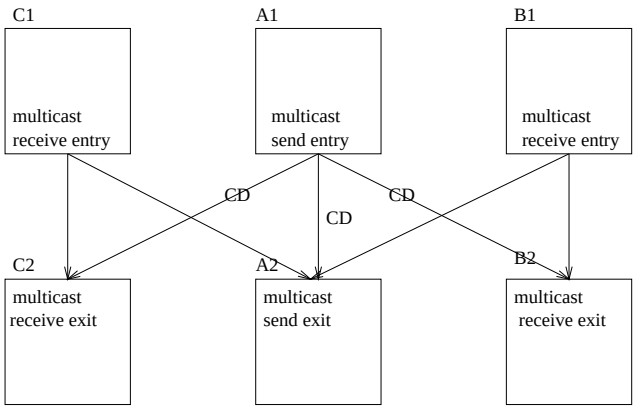


Figure 4: Mapping of Synchronous Multicast Communication

To build the precedence graph for a group of tasks involved in a multicast, each multicast send must be matched with the complementary receives of the group members. If group membership changes, then we will not be able to determine at compile-time which receives will correspond to a particular send. For instance, if the multicast group X is dynamic and if B joins the group dynamically and does a receive, when A does a multicast send to group X (and assuming we place no precedence constraints on A and B) then we do not know at compile time if B should be part of the multicast. In such cases, we will not be able to predict the behavior of the multicast and provide any guarantees, which is not suitable for real-time synchronous multicasts. To ensure predictability for synchronous multicasts, group membership must remain **static**.

3.2 Asynchronous Multicast Groups

With asynchronous messages, we provide best-effort delivery. There is no guarantee that all the processes in the group will read the message in time (although the message is sent to all the processes in the group). However, the scheduler makes a best effort to schedule the tasks so that as many processes as possible within the group will see the message in time. See Figure 5. This is done at run-time.

No pre-processing is done before run-time and no precedence is imposed on asynchronously communicating tasks, hence group membership does not have to remain static and we can have **dynamic** groups.

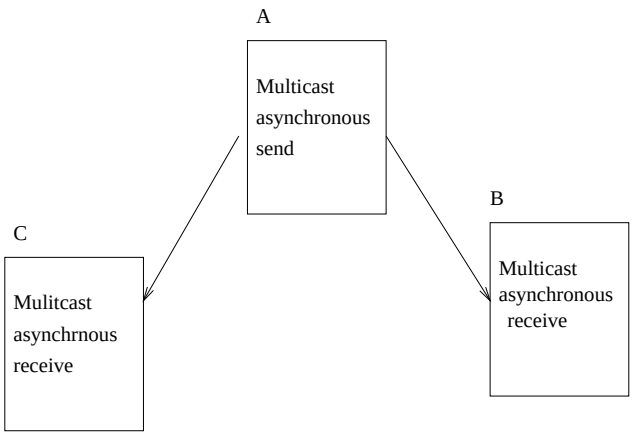


Figure 5: Mapping of Asynchronous Multicast Communication

3.3 Open Groups

Spring multicast groups are considered **open** groups because any process is allowed to send a multicast to any multicast group. Open groups do not limit multicasting to only member processes hence there is much more flexibility in the models of communication, a group need not include every possible sender which in the case of some applications would be quite inefficient. For example, client-server type communication between one of many possible processes and a group of server processes, or one way multicasts from sensor processes to a group of control processes.

3.4 Implications of Dynamic vs. Static

Traditional distributed systems most often provide dynamic groups but for a real-time operating system, dynamic groups raises a tradeoff between flexibility (dynamic groups) and predictability (static groups).

From the point of view of the programmer, dynamic groups mean that group membership is changeable at any point during the execution (because joining and leaving a group are asynchronous events). For instance, if A, B and C are process groups, but the task within C that joins C to the group, is scheduled after the send of A and before the send of B, then A and B are multicasting to the same group but with different membership. This is only known at run-time and may change with different instances of the processes. Most real-time programmers will require greater predictability than this, and hence the requirement for static groups. With static groups, the programmer knows exactly which processes are in a group. For example, if the programmer actually intends C to be part of the second multicast but not the first, the programmer can ensure this by defining two different multicast groups, one that includes C and one that does not.

Still, we provide the option for dynamic groups with asynchronously communicating process groups to give the programmer greater flexibility. The programmer could use precedence constraints, e.g., C must execute after A and B, to obtain the desired behavior. This is a trade-off of schedulability for flexibility.

Here we should stress that our notion of dynamic is restricted to the context of *one invocation* at a time.

4 Restrictions on Synchronous Multicasting

Spring processes are grouped into different types depending on timing constraints. **Critical** hard real-time processes have hard deadlines and catastrophic results if not completed on time. They are guaranteed statically. **Essential** hard real-time processes do not cause disaster on failure to meet their deadlines but have no value after that. These are guaranteed dynamically as they arrive. **Soft real-time** processes have deadlines and their values decay gradually after the deadline. These are scheduled on best-effort basis after the hard real-time processes are scheduled. **Non-realtime** processes have no deadlines and are executed when there is idle time.

Since synchronous communication produces timing constraints on the involved processes, there must be restrictions on what type of processes can synchronously multicast. A critical process cannot synchronously communicate with any processes other than critical processes because we need a static a priori guarantee. Essential processes cannot synchronously communicate with soft-real-time or non-real-time processes because the latter are not scheduled with any guarantees. Soft real-time and non-real-time processes can communicate synchronously. To enforce these restrictions, the ports and messages have been typed.

5 Handling Synchronous Communication

In section 2 we saw a simplified view of the how the task group representing synchronously communication processes is built. Here we look at it in more detail, discussing changes to the algorithms involved to accommodate multicasting.

5.1 Synchronous $1 - to - 1$ Communication

With processes involving synchronous communication, task group building is done using the *build_task_group2* algorithm [7]. If process A sends synchronously to process B, then the algorithm generates unresolved precedence relations as shown in Figure 6.

The algorithm creates the first task for A, $A1$, with the boundary at the suspension point or synchronous send call. The algorithm also creates an unresolved precedence relation ($A1$,UR_SEND($vc1$),CD($vc1$)), which notes that $A1$ is the predecessor of a yet unknown task on the other side of $vc1$, and that the relation has a delay of the magnitude of the communication delay associated with $vc1$. When processing B, the algorithm creates the first task $B1$ and generates an unresolved precedence relation ($B1$,UR_RECV(vc1),0) which means that $B1$ is the predecessor of a task representing the other side of the virtual circuit $vc1$, also yet unknown.

The *resolve_sync_comm* algorithm takes care of resolving the unresolved precedence constraints. The algorithm reads the SDL sections of the two process's executables, building two ordered set UR_A and UR_B, containing the unresolved precedence relations in the order

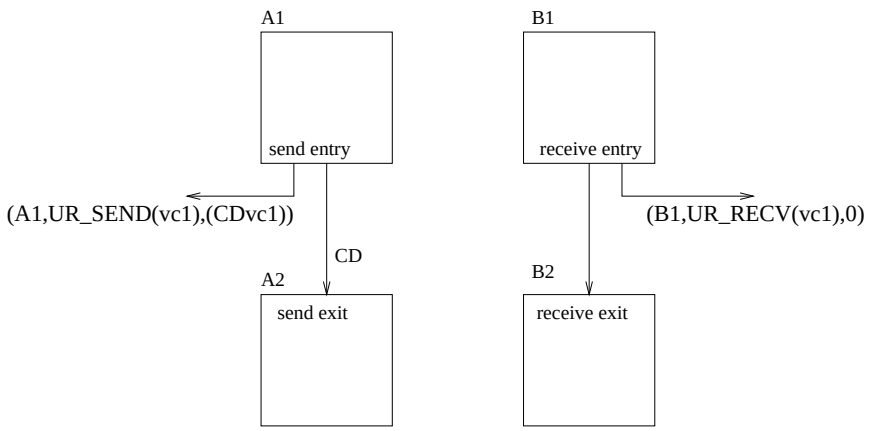


Figure 6: Unresolved precedence relations for A and B

of their listing in the task group's successor lists. An unresolved relation is of the form (*type, vc, proc, task*), where *type* indicates send or receive, *vc* is the name of the virtual circuit used, and *proc* and *task* identify the task following the suspension point.

For the previous example, UR_A and UR_B are:

$UR_A = \{(S, vc1, A, A2)\}$

$UR_B = \{(R, vc1, B, B2)\}$

The algorithm maintains a pointer for each set, identifying the next element to be resolved. The algorithm iterates, checking the current set of pointers for complementary pairs on each iteration. This being a very simple example, immediately we can see that the missing process and task fields of the *receive* in B, can be set to A and $A2$ and the missing process and task fields of the *send* in A, can be set to B and $B2$.

5.2 Synchronous Multicasts

If process A sends a synchronous multicast to process B and C, then unresolved precedence relations that should be generated are shown in Figure 7.

There should be only trivial changes to *build_task_graph2*, to generate UR_MRECV and UR_MSEND types of unresolved relations. For each multicast send suspension point, there should be n UR_MSEND precedence relations established, where n is the size of the multicast group. The *resolve_sync_comm* algorithm needs to be extended to take care of grouping complementary multicast send/receives instead of just symmetrical pairs of send/receive. The modification should not be difficult, since the concept is the same. For each current set, the algorithm iterates to find the matching *receives* and *send* that have the same virtual circuit. For instance, in the example, the ordered sets it builds are:

$UR_A = \{(S, vc1, A, A2), (S, vc1, A, A2)\}$

$UR_B = \{(R, vc1, B, B2)\}$

$UR_C = \{(R, vc1, C, C2)\}$

The missing process and task fields of the UR_MRECV in B and C, can be set to A and $A2$ and the missing process and task fields of the two UR_MSENDs in A, can be set to

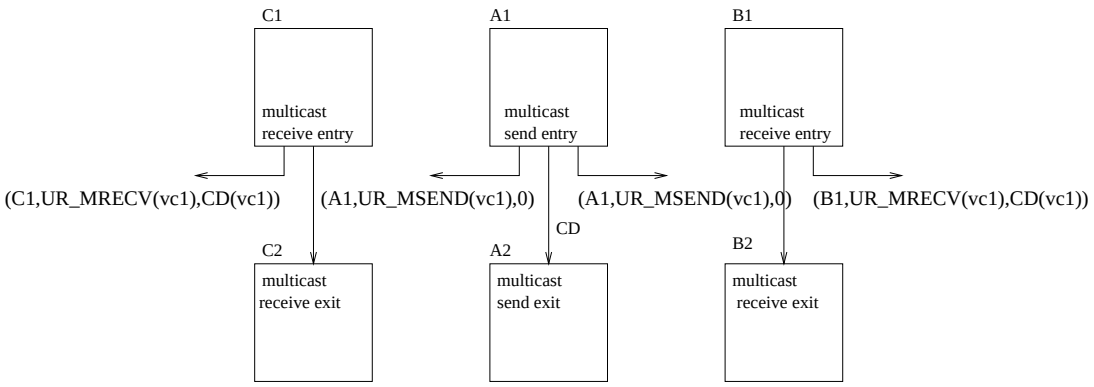


Figure 7: Unresolved precedence relations for A and B

B and $B2$ and C and $C2$.

Finally, we should note that for both synchronous $1-to-1$ and multicast communication it is possible to write applications that use the primitives in a contradictory manner (e.g. a synchronous send without a receive, or a synchronous multicast in which not all the receiving processes do the receive). In cases like this the algorithm will fail and should announce an error.

6 Proposed Implementation

6.1 Existing Spring IPC

Spring IPC is implemented using the abstraction of ports. **Ports** are kernel-protected memory objects that hold **messages**. Processes send messages by queueing and receive messages by removing messages ports. Ports are unidirectional, with bounded capacity (specified in terms of number of messages that can be queued), and have queueing and overflow policies associated with them (minimum laxity first, discard latest message, respectively). Ports are typed according to the task using them (critical, essential, soft real-time, non-realtime) and semantics of the communication (asynchronous, synchronous).

Messages are units of information passed between processes via the IPC primitives. Messages have fixed size, copy-by-value semantics and deadlines for delivery to the receiving port. They are also typed in the same ways ports are.

6.2 Extensions for Multicasting

This design is built on the current IPC subsystem which uses reflective shared memory to send and receive messages. The existing IPC mechanisms only need to be extended to provide for multicasting.

1. A multicast port is identical to a regular port **except**:

- only one message can be queued to the port
- the message is not deleted after it is read by a process, it is present until overwritten by the next message
- the message can be read by any process in the multicast group associated with that port

2. All created ports are located in SCRAMNet memory, and are allocated from a system pool. However, a multicast port belongs to the *instance* of the process group that uses it. This ensures that multiple instances of the same process group will still have unique multicast ports. This also means that process groups that multicast cannot have instances that overlap.

3. We chose to limit the multicast port queue to just one message. This is not as limiting as it may seem. Consider synchronous multicasts; since all receives in a group are scheduled after the send, there is no need to queue messages, we need only one message buffer for all receiving processes to read the message. The next send can be allowed to overwrite the message. For asynchronous multicasts, if the receive does not occur before the next send, then the previous message is lost. Hence a message is only valid from the time it was sent, until the next message is sent. If this is not acceptable, then synchronous multicasts must be used.

4. Every receiver of a multicast will make a call to create the multicast group (giving the multicast group name). If the receiver is the first to make the call, then the port is created with the multicast group name, else it is just added to the capability list for the port. Therefore, a multicast group name and its port name are synonymous. Any sending process will use the group name to discover the multicast port id.

5. During execution, any multicast group member can read its multicast port.

6. For synchronous multicast groups, members are all the processes that make the call to create the multicast group at process initialization time.

7. For asynchronous multicast groups, members are all the processes who have made the call to create the multicast port AND have executed the call to join the multicast group. If the call to join the group is placed in the initialization code of the process, then the process is a member when activated. However, if the call is placed in the main code, then joining occurs at run-time asynchronously.

8. A multicast port can be read only by processes that are in the multicast group. This is enforced by capability checking.

7 Primitives

7.1 Current IPC Primitives

Here we present a short updated version of Nahum's [4] [5] description of the primitives. We need to do this to provide updated documentation and more importantly to show that

the multicast primitives will share the same interface as existing primitives. Please see [4] and [5] for the full description of the original primitives and the pseudo-code.

Note, the following synchronous primitives `sync_send()` and `sync_recv` are pre-processed by the compiler, to obtain the correct timing graph. The compiler then replaces these calls with calls to the actual system calls, `ipc_msg_send_sync` and `ipc_msg_recv_sync`, respectively.

```
int sync_send(
   name_t *        port_name,
   port_id_t       port_id,
   msg_t *         msg
)
```

The `sync_send()` call causes the calling process to be suspended until the receiving process is passed the message. This is the call used by the programmer. The call is replaced by the SGS with a call to

```
int ipc_msg_send_sync(
   port_id_t       port_id,
   msg_t *         msg
   deadline_t      deadline
)
```

The `ipc_msg_send_sync()` call takes the passed `msg` and sends it synchronously to `port_id`.

```
int sync_recv(
   name-t *        port_name,
   port_id_t       port_id,
   msg_t *         msg
)
```

The `sync_recv()` call causes the process to suspend until a message is available. The SGS replaces this with a call to

```
int ipc_msg_recv_sync(
   port_id_t       port_id,
   msg_t *         msg
   deadline_t      deadline
)
```

The `ipc_msg_recv_sync()` call suspends until a message is available at the specified `port_id`. When a message is received, it is copied into the passed buffer, `msg`.

```
int ipc_msg_send_async(
```

```
   port_id_t       port_id,
   msg_t *         msg
   deadline_t      deadline
)
```

The ipc_msg_send_async() call takes the passed msg and sends it asynchronously to port_id. The call returns without waiting for the message to be received at the other process.

```
int ipc_msg_recv_async(
   port_id_t       port_id,
   msg_t *         msg
   deadline_t      deadline
)
```

The ipc_msg_recv_async() call checks the specified port_id to see if a message is available. If so the message is copies into msg, otherwise the call returns immediately.
Possible error codes on sending are:
(See section 7.5 for an explanation)

```
   ERROR_IPC_BAD_PORT_ID
   ERROR_IPC_MSG_TYPE_VIOLATION
   ERROR_IPC_PORT_OVERFLOW
   ERROR_IPC_PORT_UNKNOWN
```

Possible error codes on receiving are:
(See section 7.5 for an explanation)

```
   ERROR_IPC_BAD_PORT_ID
   ERROR_IPC_DEADLINE_VIOLATION
   ERROR_IPC_MESSAGE_UNAVAILABLE
   ERROR_IPC_MSG_TYPE_VIOLATION
   ERROR_IPC_PORT_UNKNOWN
```

7.2 Synchronous Multicast Primitives

7.2.1 Synchronous Send Multicast

```
int msync_send(
   name_t *        mcast_group_name,
   port_id_t       port_id,
   msg_t *         msg,
)
```

The SGS recognizes the msync_send() call as a scheduling point and the calling process is suspended at this point as a task boundary and resumed only after all processes in the multicast group corresponding to port_id have received the passed message. The msync_send() is replaced by the SGS with a call to

```
int ipc_msg_send_msync(
   port_id_t       port_id,
   msg_t *         msg,
   deadline_t    deadline
)
```

The ipc_msg_send_msync() call sends the passed message synchronously to the multicast port named port_id. This call is identical to ipc_msg_send_sync() except the port is typed as a multicast port. The sent message is copied to the kernel buffer and the call returns to the application without blocking.

Pseudo-code:

1. check that the port id is valid and convert it to a pointer to the port control structure
2. check the port and message semantic type
3. obtain a free message buffer
4. copy the passed message into the message buffer
5. this task is complete, process is suspended until next task (note, call returns in next task)

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_BAD_PORT_ID
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_OVERFLOW
ERROR_IPC_PORT_UNKNOWN
```

7.2.2 Synchronous Receive Multicast

```
int msync_recv(
   name_t *        mcast_group_name,
   port_id_t       port_id,
   msg_t *         msg,
)
```

The SGS recognizes the msync_recv() call as a scheduling point and the calling process is suspended at this point as a task boundary and resumed only after the message has been sent. This call is replaced with a call to

```
int ipc_msg_recv_msync(
   port_id_t       port_id,
   msg_t *         msg,
   ipc_flags *     flags,
)
```

The ipc_msg_recv_msync() call suspends the calling process until there is an available message. This is identical to ipc_msg_recv_sync() except the port is typed as a multicast port. The specified port is checked to see if a message is available and if so the message is copied into the passed message buffer.

Pseudo-code:

1. process is suspended, control returns in context of next task
2. check that the port id is valid and convert it to a pointer to the port control structure
3. check that the calling process is a member of the named multicast group
4. check the port and message semantic type
5. check if there is a message waiting for this process
6. if there is, copy the message into the passed buffer
7. free the message buffer

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_BAD_PORT_ID
ERROR_IPC_DEADLINE_VIOLATION
ERROR_IPC_MESSAGE_UNAVAILABLE
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_UNKNOWN
ERROR_IPC_NOT_MEMBER
```

7.3 Asynchronous Multicast Primitives

7.3.1 Asynchronous Send Multicast

```
int ipc_msg_send_masync(
   port_id_t       port_id,
   msg_t *         msg,
   deadline_t    deadline
)
```

The ipc_msg_send_masync() call sends the passed message to the multicast port named port_id. This call is identical to ipc_msg_send_async() except the port is typed as a multicast port. The sent message is copied to the kernel buffer and the call returns to the application without blocking.

Pseudo-code:

1. check that the port id is valid and convert it to a pointer to the port control structure
2. check the port and message semantic type

3. obtain a free message buffer

4. copy the passed message into the message buffer

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_BAD_PORT_ID
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_OVERFLOW
ERROR_IPC_PORT_UNKNOWN
```

7.3.2 Asynchronous Receive Multicast

```
int ipc_msg_recv_masync(
   port_id_t       port_id,
   msg_t *         msg,
   deadline_t    deadline,
   ipc_flags *     flags
)
```

The ipc_msg_recv_masync() call asynchronously checks the specified port_id for an available message. This is identical to ipc_msg_recv_async() except the port is typed as a multicast port. The specified port is checked to see if a message is available and if so the message is copied into the passed message buffer.

Pseudo-code:

1. check that the port id is valid and convert it to a pointer to the port control structure

2. check the port and message semantic type

3. check that the calling process is a member of the named multicast group

4. check if there is a message waiting

5. if there is, copy the message into the passed buffer

6. free the message buffer

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_BAD_PORT_ID
ERROR_IPC_DEADLINE_VIOLATION
ERROR_IPC_MESSAGE_UNAVAILABLE
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_UNKNOWN
ERROR_IPC_NOT_MEMBER
```

7.3.3 Asynchronous Join

```
int ipc_join_mcast_grp(
   name_t *  proc_name
   name_t *  mcast_grp_name
)
```

The ipc_join_mcast_grp() call asynchronously joins proc_name to the multicast group mcast_grp_name. This enables proc_name to receive all further asynchronous multicast messages to the group mcast_grp_name. This call must be made by each process in the process group to become a member of the asynchronous multicast group and can be made in the initialization code as well as in the main process execution code.

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_UNKNOWN
```

7.3.4 Asynchronous Leave

```
int ipc_leave_mcast_grp(
   name_t *  proc_name
   name_t *  mcast_grp_name
)
```

The ipc_leave_mcast_grp() call asynchronously removes proc_name from the multicast group mcast_grp_name. This disallows proc_name to receive any further asynchronous multicast messages intended for group mcast_grp_name.

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_MSG_TYPE_VIOLATION
ERROR_IPC_PORT_UNKNOWN
```

7.3.5 Asynchronous Query Group Members

```
int ipc_query_mcast_grp(
   name_t *   mcast_grp_name
)
```

The ipc_query_mcast_grp() call returns a list of processes that are in multicast group mcast_grp_name. (We might have to do something to consider security later).

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_PORT_UNKNOWN
```

7.4 Port Management

Only the `ipc_port_create` primitive needs to be modified. The port deletion primitive, `ipc_port_delete`, remains the same.

7.4.1 Port Creation

```
int ipc_port_create(
   name_t *           name,
   int                num,
   semantic_t         semantic_type,
   queue_policy_t     queue_policy,
   overflow_policy_t  overflow_policy,
   port_id_t *        port_id_ptr
)
```

The `ipc_port_create()` call creates a port for use in communication. This call exists, we are modifying it to allow the port to be typed as a multicast port (synchronous or asynchronous). A new port identifier is returned.

Parameters:

- `name`: unique name for port.
- `num`: size of message queue.
- `semantic_type`: asynchronous, synchronous, accept, reply, asynchronous_multicast, synchronous_multicast.
- `queue_policy`: specifies the queueing policy for the port. In the case of multicasts this is currently redundant.
- `overflow_policy`: specifies overflow policy for the port. Again, this is currently redundant for multicasts.

Possible error codes are:
(See section 7.5 for an explanation)

```
ERROR_IPC_BAD_PARAMETER
ERROR_IPC_MESSAGES_EXHAUSTED
ERROR_IPC_PORTS_EXHAUSTED
```

7.5 Brief explanation of the error codes

Possible errors occurring during sending and receiving:

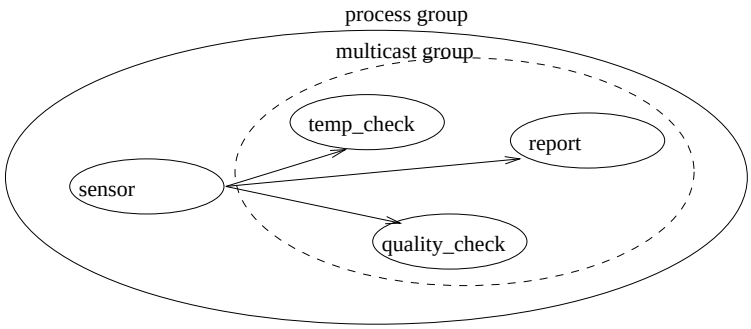
- ERROR_IPC_BAD_PORT_ID: error in port id used
- ERROR_IPC_DEADLINE_VIOLATION: message deadline is violated
- ERROR_IPC_MESSAGE_UNAVAILABLE: for synchronous only; expected message was not present
- ERROR_IPC_MSG_TYPE_VIOLATION: port semantic type and message semantic type do not match
- ERROR_IPC_NOT_MEMBER: for multicasts only; process is not a member of the multicast group associated with the port it is trying to read
- ERROR_IPC_PORT_OVERFLOW: port message queue has overflowed
- ERROR_IPC_PORT_UNKNOWN: port name is not in port registry

Possible errors occurring during port creation and management:

- ERROR_IPC_BAD_PARAMETER: bad parameters to sys_port_create
- ERROR_IPC_MESSAGES_EXHAUSTED: no free ports left in system pool
- ERROR_IPC_PORTS_EXHAUSTED: no free message buffers left in system pool

8 Code Example

Here we provide an example application use of multicasting. Consider a fictional stage of an application in a chemical manufacturing plant. First, assume we have a synchronously multicasting process group, consisting of the processes *sensor*, *temp_check*, *quality_check* and *report*. The process group is periodic; *sensor* multicasts information on the current state of the product to the other processes, *temp_check* and *quality_check* take appropriate action based on information sent by *sensor* and *report* provides a report to the human supervisor based on information sent by *sensor*. The following figure shows the process group. The multicast sender is *sensor* and the multicast group is *temp_control*, *quality_check* and *report*.



In the initialization code of each of the receivers (*temp_check_init.c*, *quality_check_init.c*, *report_init.c*) the multicast port create call must be made, to create a receiving port with 1 message buffer, for synchronous multicasting, with no queueing policy or overflow policy, and the given port id.

```
extern port_id_t mcast_port_id;
int proc_init()
{
name_t     recv_port_name = "mcast_port";
int rv;
  /* create a port for this multicast group
   */
  rv = ipc_port_create(&recv_port_name,           /* port name */
                       1,                         /* queue size */
                       semantic_type_mcast_sync,  /* semantic type */
                       0, 0,               /* no queueing or overflow policy */
                       &mcast_port_id);           /* port id */
...
```

In *sensor_init.c*, the port name must be looked up

```
extern port_id_t mcast_port_id;
int proc_init()
{
int rv;
  /* find the addresses of the multicast port that I'll be sending to
   */
  rv = ipc_port_name_lookup((name_t *)"mcast_port", &mcast_port_id);
...
```

Within *sensor.c* the call to send a synchronous multicast will occur

```
extern port_id_t mcast_port_id;
proc_exec()
{
  mcast_msg_t * mcastp;             /* template for multicast message */
  msg_buffer_t mcast_send_buffer;   /* IPC message buffer             */
...
 /* multicast sensor data
  */
  mcastp = (mcast_msg_t *) &mcast_send_buffer;
  mcastp->data = data;
  msync_send("mcast_port", mcast_port_id, &mcast_send_buffer);
...
```

And finally, within *temp_check.c*, *quality_check.c* and *report.c*, the synchronous multicast is received.

```
extern port_id_t mcast_port_id;
proc_exec()
```

```
{
  mcast_msg_t * mcastp;             /* template for multicast message */
  msg_buffer_t mcast_recv_buffer;   /* IPC message buffer             */
...
 /* receive the multicast and obtain the sensor data from it
  */
  msync_recv("mcast_port", mcast_port_id, &mcast_recv_buffer);
  mcastp = (mcast_msg_t *) &mcast_recv_buffer;
  data = mcastp->data;
...
```

Just for the sake of an example, if the multicast were asynchronous, then

- the msync_recv is changed to ipc_msg_recv_masync
- the msync_send call is changed to ipc_msg_send_masync
- In the initialization code of each of the receivers (*temp_check_init.c*, *quality_check_init.c*, *report_init.c*) the multicast port create call is made but with asynchronous multicast semantic type, and the call to join the multicast group is added.

```
extern port_id_t mcast_port_id;
int proc_init()
{
name_t     recv_port_name = "mcast_port";
int rv;
  /* create a port for this multicast group
   */
  rv = ipc_port_create(&recv_port_name,           /* port name */
                       1,                         /* queue size */
                       semantic_type_mcast_sync,  /* semantic type */
                       0, 0,              /* no queueing or overflow policy */
                       &mcast_port_id);           /* port id */
  rv = ipc_join_mcast_group(this_process_name, mcast_port);
...
```

Note, if the call to join is done in the proc_exec() instead of in the proc_init(), then the process is not a member of the multicast group until the call is executed at run-time.

9 Summary and Status

This report has outlined the need for a multicast mechanism in Spring for $1 - to - N$ communication in process groups. We have defined the semantics for such communication, covering both synchronous and asynchronous multicasts. Finally, we have provided a design for an implementation based on current Spring IPC primitives. The actual implementation work is the next phase.

References

[1] Birman, P., Cooper, R. and Gleeson, B. "Design Alternatives for Process Group Membership and Multicast," Technical Report, Cornell University, Ithaca, NY, Dec. 1991.

[2] Joseph, T. and Birman, K. "Reliable Broadcast Protocols," Technical Report, Cornell University, Ithaca, NY, June 1988.

[3] Kandlur, D., Kiskis, D. and Shin, K. "HARTOS: A Distributed Real-Time Operating System," *IEEE Software*, Sept. 1992.

[4] Nahum, E., "Semantics of Spring IPC," Spring Project Documentation, University of Massachusetts, Amherst, MA, August 1990.

[5] Nahum, E., Ramamritham, K. and Stankovic, J. "Real-Time Interprocess Communication in the Spring Kernel," Spring Project Documentation, University of Massachusetts, Amherst, MA, May 1992.

[6] Niehaus, D., "Spring System Calls," Spring Project Documentation, University of Massachusetts, Amherst, MA, June 1992.

[7] Niehaus, D., "Program Representation and Translation for Predictable Real-Time Systems," Ph.D. Dissertation, University of Massachusetts, Amherst, MA, Feb. 1994.

[8] van Renesse, R., Hickey, T., and Birman, P. "Design and Performance of Horus: A Lightweight Group Communications System," Technical Report, Cornell University, Ithaca, NY, 1994.

[9] Stankovic, J. and Ramamritham, K. "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, 8(3):62-72, May 1991.

[10] Tanenbaum, A. *Modern Operating Systems*, Prentice-Hall, 1992, pp445-460.