

UW-API: A Network Routing Application Programmer's Interface (Draft version 1.2)

Parmesh Ramanathan, Kewal Saluja, Kuang-Ching Wang, Thomas Clouqueur
Contact: parmesh@ece.wisc.edu
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
October 29, 2001

1. Introduction

This document describes University of Wisconsin-Madison's application programmer's interface (UW-API) for sensor network communication. UW-API has six core primitives supporting basic message communication and collaborative computation. More primitives may be included in later versions. A detect-and-track example using UW-API is described in the document to illustrate the usage of UW-API in simple collaborative signal processing applications.

2. The API

The UW-API is motivated by the well-known message-passing interface standard MPI-1.1 proposed by the MPI forum in 1995. MPI-1.1 was proposed for distributed computations in parallel computer systems and networks of workstations. In contrast, the UW-API is tailored for distributed computations in sensor networks.

The UW-API views a sensor network at two different granularities. Some primitives in the UW-API are intended for invocation by a single sensor node while the others are intended for simultaneous invocation by a group of sensor nodes located in a certain *geographic region*. Overall, the computation is *region-based*, i.e. all operations consider a region of nodes instead of any single node to be their sources or targets. The need for *region-based* operations will become clear when we formulate the detect-and-track example. A region is a logical entity grouping a set of sensor nodes within a specified geographical range. Moreover, a region is an atomic unit in all UW-API primitives. Besides geographical range, regions can be defined in association with specific types of sensor nodes to carry out different tasks. Overall, regions must be created before any computation using UW-API, each region is associated to a geographical range and certain sensor types, and multiple regions can be created with overlapping or non-overlapping parameters, such as their ranges or the associated sensor types.

Two region management primitives, `SN_CreateRegion` and `SN_DeleteRegion`, are defined to respectively create and delete regions by specifying their geographical coordinates and their member sensor types. Three primitives, namely `SN_Send`, `SN_Recv`, and `SN_Reduce`, are for exchange of information between the sensor nodes. `SN_Barrier` provides a synchronizing mechanism for nodes in the same region.

2.1 SN_CreateRegion

The prototype of the primitive is as follows.

```
RegionIDType RegionID =
SN_CreateRegion( RegionIDType SourceRegion,
                 RangeStruct Range,
                 RangeStruct ManagerRange,
                 INT SensorType,
                 INT Timeout)
```

`SN_CreateRegion` creates a region to refer to a group of nodes equipped with the specified `SensorType` in the given geographical `Range`. A `RegionID` is returned upon success for future reference. `SourceRegion` indicates the ID of the region in which the calling node belongs. `RangeStruct` is defined as a four-element array of type `RegionCoordType` (now defined to be `short` in C) indicating the coordinates of the upper-left and lower-right corners of a rectangular area. A `ManagerRange` is a defined range encompassing nodes that will serve as leaders for collaborative computations in the region, such as `SN_Reduce`, or act as a point of contact for the region. The `Timeout` argument specifies the maximum time in seconds the call should return. If the region is not successfully created within `Timeout` seconds, the call returns `FAILURE`.

A successful `SN_CreateRegion` notifies all nodes in the created region of their membership in this new region. A node can belong to multiple regions. Multiple regions are handled with the multithread-programming model, which will be addressed later. Nodes in the manager region reply an `Ack` to the region of the caller. `SN_CreateRegion` returns the `RegionID` of the new region upon successful reception of an `Ack`. The `RegionID` is a 16-bit positive integer generated at the caller with an 8-bit random number as its high byte plus an 8-bit timestamp as its low byte.

The region manager notion is the equivalence to the root process in a distributed computing paradigm. Collective operations among several processes are often realized hierarchically with one being the coordinator and all others being slaves. In MPI 1.1, collective operations such as synchronization, message collection, broadcast, data reduction, and so on, specify a root process in each call. In UW-API, a region manager is defined to coordinate all collective operations among nodes within the region. Furthermore, in a *region-based* computing model, communication among regions is needed. Though not a must, region managers can be programmed to be the only place where inter-region messages are sent and received, thus reducing redundant messages and ensuring message consistency. The *region-based* model considers a region instead of a node as the smallest logic entity. Hence, a region manager is also defined as a range that can be a subset of more than one node in the region. As we incorporate fault-tolerance later, multiple nodes serving as region managers also avoid any single point of failure.

2.2 SN_DeleteRegion

SN_DeleteRegion destroys a previously created region with its handle. The prototype of the primitive is as follows.

```
INT Status = SN_DeleteRegion( RegionIDType SourceRegion,  
                             RegionIDType RegionID,  
                             INT Timeout)
```

RegionID is the handle of the region to be deleted. Nodes in the deleted region will be notified and the region manager nodes reply an Ack to the SourceRegion in which the caller resides. The call returns SUCCESS upon successfully receiving an Ack. The Timeout argument specifies the maximum time in seconds the call should return. If the region is not successfully deleted within Timeout seconds, the call returns FAILURE.

2.3 SN_Send

In MPI-1.1, MPI_Send is a point-to-point communication primitive that sends information from a source node to a destination node. In sensor networks, however, commands or information are meant for regions instead of any specific node. Therefore, we define a point-to-region communication primitive, SN_Send, which is called at a node to send information to all nodes in a destination region. The prototype of the primitive is as follows.

```
INT Status = SN_Send(   RegionIDType SourceRegion,  
                       RegionIDType DestinationRegion,  
                       DATATYPE* Data, INT Datatype, INT Size,  
                       TAGTYPE Tag)
```

The SourceRegion indicates the region of the sending node and the DestinationRegion specifies the target. The need to specify the source region comes with the fact that a node can potentially belong to multiple partially overlapped regions and the SourceRegion indicates the particular region it represents when sending the message. Data buffer stores the message to be sent, which is of Datatype and length Size in bytes. Current supported types are STR (char string in C), T_INT, T_FLOAT, T_SHORT, T_DOUBLE, T_LONG, and a special command message SPAWN_THREAD, which is used to spawn a thread in the destination region. A Tag is included to indicate the type of information sent, such as a command (CMD) or data (DATA), or the name of the thread to be spawned (DETECT_AND_TRACK_THREAD). SN_Send hands the message to the routing agent and returns. A Status indicator is returned as SUCCESS or FAILURE; a FAILURE occurs only if the system resource runs out and refuses to service the call.

2.4 SN_Recv

In MPI-1.1, receiving a packet can be either blocking (MPI_Recv) or non-blocking (MPI_Irecv). A blocking receive returns only after it successfully gets a message with a matching tag in the allocated buffer, while a non-blocking one returns immediately after

allocating a buffer space. Whether or not the message has arrived in the buffer for a non-blocking receive call relies on the polling function `MPI_Test` to find out.

Unlike networks of workstations, sensor networks have asynchronous computing patterns and unreliable communication. A blocking receive can indefinitely halt a node's operation while waiting for a lost or out-of-order message. Nevertheless, some sensor-network applications do expect to use a blocking receive if subsequent operations is dependent on some expected message. To avoid an indefinite halt while providing a blocking receiving functionality, we propose an `SN_Recv` that blocks for a user-specified time period receiving a desired message. An `SN_Recv` with zero timeout mimics a non-blocking receive while one with a finite timeout will block the instruction flow until the message has arrived or the timeout is reached. Since an `SN_Recv` can timeout without successfully receiving a message, the programmer is responsible to check for the returned status of the call. The prototype of the primitive is as follows.

```
INT Status = SN_Recv(   RegionIDType SourceRegion,
                       DATATYPE* Data, INT Datatype, INT Size,
                       TAGTYPE Tag, INT Timeout, void* arg)
```

The arguments specify the `SourceRegion` a node wants to receive a message from, the `Data` buffer of `DATATYPE` being `INT`, `FLOAT`, `BINARY`, etc, with a length of `Count`. The `Tag` identifies the expected message type, such as `CMD` or `DATA`. The maximum time the call blocks is specified with `Timeout` in seconds. The `SourceRegion` and the `Tag` can be specified with the wildcard `ANY` to receive packets from any region and/or with any tag. Within `Timeout` seconds, `SN_Recv` puts the first message received with a matching tag in the data buffer. If no matching message is received within `Timeout` seconds, the call returns without filling the buffer. A `Status` indicator is returned as `SUCCESS` or `FAILURE`. The `void arg` pointer is a required data structure when working in a multi-threaded environment. Multi-threaded programming will be detailed in later sections.

2.5 SN_Reduce

In MPI-1.1, `MPI_Reduce` is a collective communication primitive that collects data from all nodes in a specified group and performs an operation such as sum, min, or max on all collected data. A similar primitive in UW-API is `SN_Reduce`. `SN_Reduce` is a collective operation that needs to be called by all nodes in a region when needed. Nodes in the manager region collect data from all other nodes in the whole region and perform the specified operation which can be logical operations such as `AND`, `OR`, etc., or arithmetic operations such as `SUM`, `MEAN`, etc. Currently supported operations are `MAX`, `MIN`, `SUM`, and `MEAN`. The prototype of the primitive is as follows.

```
INT Status = SN_Reduce( RegionIDType Region,
                       DATATYPE* Data, INT Datatype, INT Count,
                       OPTYPE OPERATION, INT* numnodes,
                       INT Timeout, TAGTYPE Tag, void* arg)
```

The result of the operation will be stored in-place in the `Data` buffer at nodes in the manager region. `SN_Reduce` is a blocking call. Two criteria define the return condition. The call returns either when the number of nodes that responded to the manager exceeds `numnodes`, or `Timeout` seconds have elapsed. A `Status` indicator is returned as `SUCCESS` or `FAILURE`. The void arg pointer is a required data structure when working in a multi-threaded environment. Multi-threaded programming will be detailed in later sections.

2.6 SN_Barrier

Barrier is an important primitive in distributed computing for several processes to synchronize at a certain point in a program. UW-API offers the `SN_Barrier` call that synchronizes the operation of multiple nodes in the specified region. When a node calls `SN_Barrier`, it is blocked until a user-specified number of nodes have executed this call, such that these calling nodes are synchronized with their operations at the time `SN_Barrier` returns. Programmers are responsible to properly design their flow of operations such that the `SN_Barrier` they use are reachable by all nodes that are expected to synchronize, such that synchronization can eventually be reached.

The prototype of the primitive is as follows.

```
INT Status = SN_Barrier(RegionIDType Region, INT numnodes, INT Timeout,
                       TAGTYPE Tag, void* arg)
```

All nodes belonging to `Region` should all call `SN_Barrier`, and the call returns when `numnodes` nodes have done so. To avoid indefinite blocking, a `Timeout` in seconds is also given. A `Status` indicator is returned as `SUCCESS` or `FAILURE`.

3. Multi-threaded programming model

4. A Detect-and-Track Example

Figure 1 shows a sensor field deployed in a grid topology. Sensors along the periphery are assumed to have classifying capabilities. We assume that once any hostile vehicle passes by these sensors, it can be correctly detected. Sensors without such classifying capabilities rely on tracking algorithms to detect/classify the vehicle based on its previous detection history. Henceforth, once an intruding vehicle gets detected on the periphery, the detection information should be passed on to other sensors that will potentially detect the vehicle soon. The example given illustrates how the command and control center (CCC) initiates a detect-and-track query for a TEL, and how the sensor nodes collaboratively detect and continuously track the TEL's location in the sensor field with a *region-based* approach.

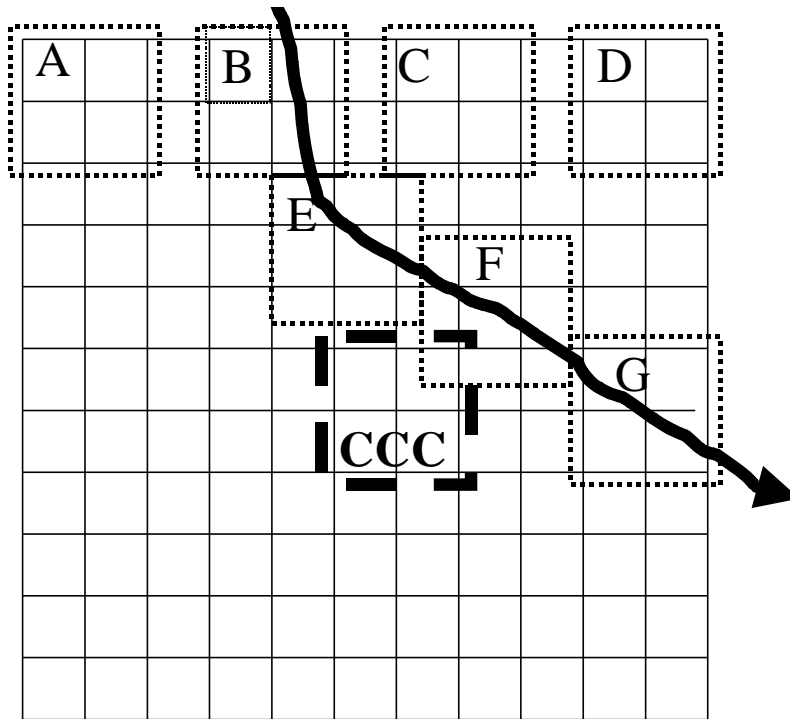


Figure.1. Sensor nodes deployed over grid points. Regions are explicitly created to group sensor nodes with same functionalities or to perform collaborative tasks.

Without loss of generality, we assume the TEL is to enter the field somewhere along the north edge. To start with, CCC creates four regions to group the classifying sensors on the north edge, shown as regions A to D in Figure 1. To start the operation, CCC gives out four separate DETECT_TRACK commands to nodes in regions A to D that essentially say “Detect and track any TEL in the field.” All nodes in regions A to D will receive the command and start detection. Once a TEL is detected, say in region B, region B will estimate the future passage of the TEL and create one or more new regions that cover the potential area the TEL is about to enter. Region B will task the newly created regions with the same DETECT_TRACK commands and send them necessary detection information. Such a detect-and-create process is repeatedly carried out along the TEL’s passage until it leaves the field or certain actions are taken to remove it. Based on necessity, regions that detect the TEL can also send the detection result to the originating region, which is the CCC, for monitoring purposes.

Within each region, sensor nodes collaborate to detect and track the target. The manager of each region, which is itself a region, coordinates collaborative operations such as SN_Reduce. Each node is sensing independently; the synchronizing primitive SN_Barrier is used to ensure all nodes in the region have completed their measurements before the manager collects their results. Using SN_Reduce, the manager identifies the maximum sensor reading among all nodes in the region. Based on SN_Reduce’s result, the manager makes the final decision whether a detection of TEL is positive. To continuously track the TEL, the manager also creates more new regions along the TEL’s

estimated passage, task them with DETECT_TRACK commands, and send them necessary detection information.

Figure 2 highlights the key message exchanges between sensor nodes using UW-API. In this figure, each vertical line shows the progress time at a corresponding sensor node in the specified region. A pseudocode for this example is given below.

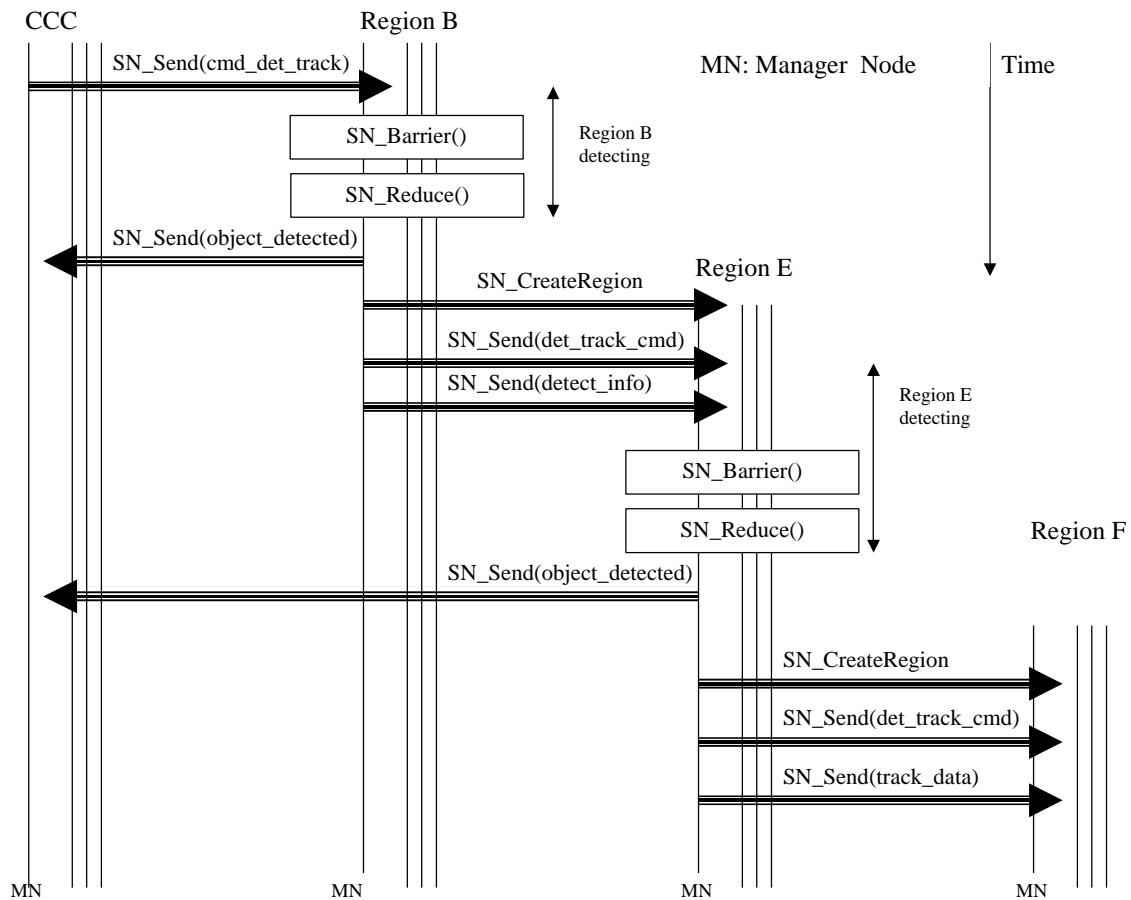


Figure.2. A snapshot of the messages exchanged between regions A, B, E, and F in the detect-and-track example.

CCC initiates the track_and_detect query with the following procedure:

CCC

```

Void main()
{
    INT Msg;

    Region_A=SN_CreateRegion([x1 Y1 x2 Y2], [x3 Y3 x4 Y4], ANY, 10);
    Region_B=SN_CreateRegion([x5 Y5 x6 Y6], [x7 Y7 x8 Y8], ANY, 10);
    Region_C=SN_CreateRegion([x9 Y9 x10 Y10], [x11 Y11 x12 Y12], ANY, 10);
    Region_D=SN_CreateRegion([x13 Y13 x14 Y14],[x15 Y15 x16 Y16], ANY, 10);

    Msg = CMD_DETECT_TRACK_TEL;

    SN_Send(My_Region, Region_A, &Msg, INT, 1, CMD);
    SN_Send(My_Region, Region_B, &Msg, INT, 1, CMD);
    SN_Send(My_Region, Region_C, &Msg, INT, 1, CMD);
    SN_Send(My_Region, Region_D, &Msg, INT, 1, CMD);

    While(SN_Recv(ANY, &Msg, INT, 1, DATA, 0))
        log_detection(&Msg);
}

```

Command thread for sensor nodes

```

Void main()
{
while(TRUE)
{
    if(SN_Recv(ANY, &command, INT, 1, CMD, 0)) // CMD stands for command
    {
        Switch(command)
        {
            case CMD_DETECT_TRACK_TEL:
                pthread_t *thread;
                RegionType *SourceRegion;
                GetSourceRegion(&SourceRegion);
                pthread_create(thread, NULL, &detect_track, &SourceRegion);
                break;

            default:
                break;
        }
    }
}
}

```

Detection thread for Sensor Nodes

```

Void detect_track(RegionType *SourceRegion)
{
    INT    Msg;           //One integer coded message
    INT    success, retry;
    FLOAT  INFO[10];     //One floating point array, storing detection information

    init();              // initialization procedures

    success=0;
    retry=0;
    while(success){
        if(retry<10){
            SN_Recv(SourceRegion, &INFO, FLOAT, 10, DATA, 10));

```

```

        Retry++;
    }
}
confidence=detect(TEL);
SN_Barrier(MyRegion, 10, 10);
SN_Reduce(MY_REGION, confidence, INT, 1, MAX, 10);

If(confidence<THRESHOLD){
    Msg = NOT_DETECTED;
    SN_Send(My_Region, CCC, &Msg, INT, 1, DATA);
}
else{
    if I am in manager region
    {
        RangeStruct NextRange=estimatePassage();
        RangeStruct ManagerRange=getManager(NextRange);
        NewRegion=SN_CreateRegion(NextRange, ManagerRange,ANY, 10);

        Msg = TEL_DETECTED;
        SN_Send(My_Region, CCC, &Msg, INT, 1, DATA);
        Msg = CMD_DETECT_TRACK_TEL;
        SN_Send(My_Region, NewRegion, &Msg, INT, 1, CMD);
        GetDetectionInfo(&INFO);
        SN_Send(My_Region, NewRegion, &INFO, FLOAT, 10, DATA);
    }
}
}
}

```

NOTES:

1. The example code is assumed to be executed concurrently at all sensor nodes.
2. The UW-API can be implemented on top of existing sensor network routing protocols, such as diffusion routing proposed by USC/ISI. However, we are currently in the process of exploring and evaluating a new routing protocol, which we believe is more efficient for implementing this API.
3. We are also evaluating the ease of implementing collaborative signal processing algorithms using UW-API and the USC/ISI/MIT/LL network routing API.