Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

A Self-Configuring Distributed Kernel for Satellite Networks

Scott Cannon and Larry Denys
Dept. of Computer Science, Space Software Laboratory
Utah State University, Logan, UT., USA 84322-4205

Abstract. The Space Software Laboratory is developing a self-configuring distributed kernel to be used on future satellite missions. The completion of this system will allow a network of heterogeneous processor nodes to communicate and broadcast in a scalable, self-configuring manner. Node applications software will be transparent to the underlying network architecture, message routing, and number of network nodes. Nodes may halt or be reset and later rejoin the network. The kernel will support in-flight programming of individual nodes.

1. Introduction

In small, low-power satellites, processing is often distributed. Developers find it cost and power efficient to do processing and control close to each of many data sources. As such, each sensor or experiment is typically controlled by its own small processor. This approach also speeds development and reduces cost by parallelizing development and testing efforts. The availability of micro-miniaturised space-tested processors [1,2] with extremely low power consumption has made this approach mass, size, and power efficient.

In such a networked system, each experiment or sensor node typically communicates data and receives commands from a satellite control or data handling computer that manages communication with the rest of the satellite and the ground telemetry system. Communication bandwidth requirements are often modest.

In many missions, it is important for a sensor or experiment node to be situationally aware - important measurements made by one node must be communicated to other nodes in order to take advantage of the collective capability of the entire payload. For example, one sensor may be capable of detecting a solar flare event. If such occurs, other experiments or sensors should be notified of this important change in environment to allow them to adjust operational parameters.

During flight, nodes may be powered off, reprogrammed, or otherwise leave and join a network. In some important cases, it has even been necessary to add new sensor or experiment nodes to a satellite system just prior to launch. In the future, missions are envisioned where new nodes will be added or docked to a satellite network in orbit. In such a system, sensor nodes would be developed independent of the rest of the network or even the satellite mission. Nodes would become standardized off-the-shelf modules for rapid and cost-effective satellite missions.

Unfortunately, this type of communication often requires the capabilities of each node, the network configuration, and message routing to be fixed very early in the satellite development cycle. Changes to this fixed configuration may result in significant time and budget expenditures in order to adapt and reprogram nodes.

Historically, such satellite network software systems have been custom developments. A significant amount of development time and budget is spent on specifying and developing network communications software and messaging protocols.

To address these satellite development issues, the Space Software Laboratory of Utah State University is developing a self-configuring distributed communications kernel. The goal is to allow an experiment or sensor node to be developed independent of the rest of the network. When a node is added to or leaves the network, the system will self-configure communications. The system will allow a new node to become situationally aware even though it is not aware of specific network members or configuration. It will also allow all other nodes to take advantage of the sensor capabilities of the new node without software modifications. This combined capability is often called *plug & play* in the satellite community. In-situ reprogramming of nodes will also be supported.

2. Requirements

The kernel will allow the ground station to upload and execute new software on a specified network node. In addition, the kernel will provide a set of interface functions to the application programmer of a satellite node to hide a) details of communications and routing and b) the current network situation. The kernel requires a simple two-thread scheduler to provide a CSP channel model for process blocking and local process messaging. Our intent is to use an existing operating system for local process scheduling and channels. The kernel will simply add a self-configuring network for inter-node network communications. The basic requirements of a self-configuring network are the following;

2.1 Functional Flexibility.

Applications software on any sensor or experiment node must be upgradable in flight. For flexibility, in-situ programming must be provided in a complete and general high-level language (such as C or C++).

2.2 Fault-Tolerance.

A halt failure of a sensor or experiment node must not halt or impair the continued operation of the rest of the network. Reset or restarted nodes must begin again to participate in the network.

2.3 Dynamic Scalability.

Network system software must be self-configuring for additions, removals, and failures of sensor or experiment nodes. Application software for one node must not require details of the architecture, network configuration, or network members to keep aware of the global situation.

2.4 Standardized Interfacing.

The programmers for sensor or experiment nodes will require only minimal understanding of the communications system and network operation. Space tested off-the-shelf components must be used. It must be possible to independently develop and test sensor or experiment nodes.

2.5 Scalability.

No software modifications or hardware capabilities, routers, switches, or ports will be required for the addition of a node (other than that required by the node itself).

3. Design

Much like a LAN, the applications software for each node will sit atop a distributed network kernel, which provides communications and overall operations control. Network communications will appear to be bus-oriented. On the other hand, the satellite control computer will communicate with the network as if it were a single node. In other words, the satellite will not need to be aware of network size or configuration (fig. 1).

In the following discussion, the use of the kernel will first be explained from the perspective of the applications programmer for a given node. Next, the paper will discuss the implementation details of the kernel design itself.

For space and execution efficiency as well as program development support, the kernel is being implemented in C++.

3.1 Self-Configuring Communications.

For simplicity and flexibility, all application code will communicate only with the kernel. The kernel appears to be a central repository of status and situational information. The applications software developer for a node may utilize a set of library functions for kernel interaction: These library functions (described below in symbolic code) provide support for configuration-independent communications.

The kernel design utilizes the bulletin board model similar in concept to the Linda or tuple-space network operating system family [3-7]: Imagine a central room with a set of bulletin boards – one board for each network node. Each node may post information to its own bulletin board and read the information from another's. One additional bulletin board is global - any node may AND or OR information on this board to update the global situation. The kernel library is defined with the functions outlined in the sections below.

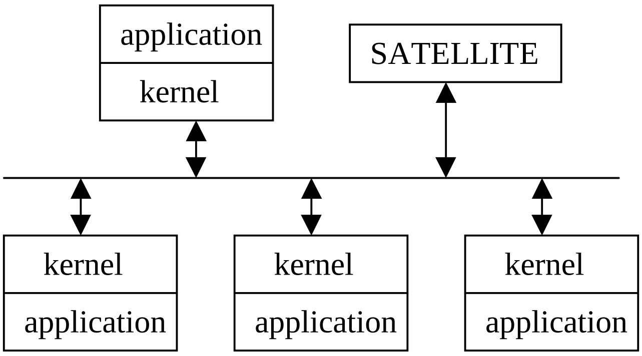

Figure 1. Sensor node configuration

3.1.1 Registration

A node joins the network by registering the `names' of its sensors and experiments with the network using the following library function:

```
int Register (string sensor_name);
```

Each registering sensor is assigned a bulletin board with its ‘name’ at the top. Each bulletin board has space for a status word and data messages. Sensor names are hierarchical. For example, two different sensors that both detect solar flares may begin their names with the prefix "SF". A node is expected to register all its experiments and sensors upon booting up.

A node may also request a list of all currently registered sensors or experiments;

```
int WhoIsRegistered (list& sensor_list);
```

3.1.2 Sensor Status Information

A sensor or experiment may posts its status to its system bulletin board with the following library function;

```
int PostStatus (string sensor_name, int sensor_status);
```

The system identifies the appropriate board to post this information. The sensor status is simply an integer – the definition and meaning of status values or bits can be defined by the sensor developer. If this sensor name is not registered, an error is returned. If this sensor name was registered by another node, an error is returned and the call has no effect.

A sensor node may also request the status of another sensor or experiment by referring to its name;

```
int GetStatus (sensor_name, int& sensor_status);
```

The system identifies the appropriate board to read and return this information. If the sensor is not registered, an error is returned.

Since names are hierarchical, an application can infer the function of another sensor from its name. For example, if the application software on a node would like to be aware of the occurrence of a solar flare (but is not itself capable of such detection), it can search the list of registered sensors for any with the name prefix "SF" and then retrieve the current status of such available sensors.

3.1.3 Situational Information

The kernel also maintains a separate bulletin board with a word of global situation information. This is a string of bits that may be manipulated by any node with the operators AND, OR, and EOR (exclusive OR).

```
int PostSituation (string sensor_name, int situation,
     Operator operator_name);
```

Any application may retrieve this global situation;

```
int GetSituation (int& situation);
```

Each bit of this global word is pre-assigned to a particular situation. Situations reflect states of nature that are detected by sensors or measured by experiments and as such, are independent of sensors. For example, any sensor node that detects a solar flare will OR bit three of the situation word. Any node wishing to be aware of a detected solar flare will retrieve the global situation word and examine bit three. The examining node need not be aware of which sensor node or nodes set this bit - only that if a sensor node capable of detecting a solar flare is on-board at the moment, it will set this particular bit when a flare is evident. The examining node can deduce that a zero in this bit may mean that a flare is not evident, or that no sensor is on board capable of detecting the flare. No provision is made for conflicting sensors.

For more real-time operations, an application node may request that a certain function or procedure automatically be invoked each time a given situation occurs;

```
RegisterForInterrupt (void *function_name,
     int situation_mask);
```

In this library function call, the function to be automatically invoked is indicated by the first argument. The second argument is an integer mask to be AND'd with the current situation. Any time this AND operation produces a true result, the indicated function is invoked on the node making this call.

3.1.4 Data Transfer

Data messages may also be posted on a bulletin board for access by other nodes. (The details of hand shaking, polling, etc. are all hidden from the programmer.)

```
int PostData (string sensor_name, Data data_message);
```

If the sensor is not registered or was registered by another node, an error is returned. The most recent data message posted by another sensor may be read with the following call;

```
int ReadData (string sensor_name, Data& data_message);
```

If the sensor is not registered, an error is returned. Data to be stored for future satellite downlink telemetry is stored with the following call. Normally, telemetry data is not available to other nodes.

```
int TelemetryData (string sensor_name, Telemetry
     message);
```

3.2 Implementation

The above discussion shows how the kernel would be utilised from the application programmer's perspective. The discussion below outlines how the kernel itself is implemented.

The distributed kernel is divided into four functional units that are replicated on each sensor or experiment node:

- Physical communications layer -- responsible for physical network communications and protocols.
- Network layer - responsible for formatting and handling communications at the message level.
- Interface layer -- a standard library of function calls available for applications to perform communications, data transfer, and scheduling.
- Program control -- accepts and executes new applications software from the data handler.

The first three units form a simple protocol stack, allowing the details of physical communications as well as handshaking and messaging protocols to be separated from the application program. The physical layer and program control layers are specific to a given architecture. In a heterogeneous system, each node's kernel will have a physical and control unit specific to that architecture. The interface and network layers are portable. The design of each of these four functional units is described below:

3.2.1 Physical Layer.

Many different connection strategies could be employed including Mil. Std. 1553 and the newly proposed European Space Agency *SpaceWire* standard. The first version of the kernel system is targeted at small, low-power, scalable satellite needs and utilises RS422 configured as a half-duplex multi-drop differential bus. Low-voltage differential signalling associated with 422 provides a robust connection and no special router or port support is required. As a result, *plug & play* is easily achieved. Simple space-tested RS422 bus drivers are readily available and interface easily to a processor serial port. (To be bus compatible, a processor must enable the transmit signals. In other words, control when the node is physically connected to the transmit bus and able to transmit.). Each node requires a single serial port interface (transmit and receive), a single-bit transmit-enable discrete (transmit enable), and a single- bit input (Satellite RTS) (fig. 2).

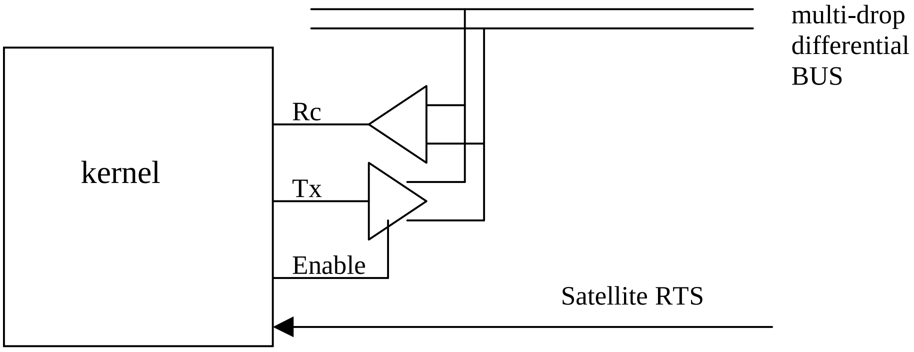

Figure 2. Physical network layer using RS422 in a half-duplex multi-drop bus.

To avoid contention, one node serves as the bus master. Each node may only transmit when polled by the master to eliminate the need for special contention circuitry.

The satellite control computer interfaces to the sensor/experiment network using a similar RS422 serial port (fig. 3).

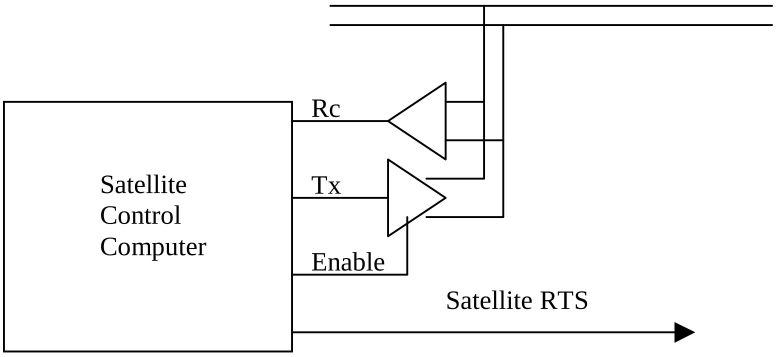

Figure 3. Satellite control computer connection to the network.

The satellite control computer does not need to recognize the network configuration or bus master. It simply treats the network as a single kernel and does not participate in the kernel master/slave protocol. To transmit a message, the satellite control a) asserts the RTS (request to send) discrete, b) pauses, c) transmits a message, and d) receives the reply(s). It is the responsibility of the network master kernel to recognize the satellite control computer request by monitoring the RTS discrete. The pause allows the master kernel to complete any current messaging and turn the bus over to the satellite. When communicating with the network, the satellite control computer simple ignores the serial port interface and leaves RTS de-asserted.

3.2.2 Network Layer.

At the network layer, messages are formatted and routed to the local kernel or application. A network message consists of an ASCII SOH byte followed by a command byte and data as diagrammed in table 1. If a message contains a SOH byte as data, this byte is coded twice. In other words, a single SOH indicates the beginning of a network message while a double SOH indicates a single data value.

Table 1. Network messages

Message	**Purpose**
SOH, *node_id*	poll message from the master node to a slave node ID
SOH, 0x80, *node_id*	end-of-file message response
SOH, 0x81, *length*, *data*	data message response (0...255 bytes) from a slave node
SOH, 0x82, *node_id*	coup notice from the new bus master node
SOH, 0x83	data dump request from the satellite
SOH, 0x84, *node_id*	data dump request from the master to a slave node ID
SOH, 0x85, node_id, length, data	application-specific command or control message from the satellite to a specific node ID

The first three messages are associated with normal network operation. Each experiment or sensor node developer is assigned a unique ID when a sensor or experiment is first commissioned (1..255 excluding SOH). The master node transmits a poll message to each current slave node in a round-robin manner and awaits a response. The network layer of each slave node listens for a poll referencing its ID that establishes a logical connection to the master. The slave kernel responds to the poll with a data message or an end-of-file message. If the slave returns a data message, the master responds with a reply data message and awaits another data message from the slave.

If the slave responds with an end-of-file message, the master polls the next slave ID to establish a connection with the next node. Naturally, the master node may have its own sensors and experiments and must logically poll itself as well.

This approach is similar to the Mil. Std. 1553 protocol. A node ID is independent of the sensor ‘name’ mentioned above and is not related to the node purpose or use. In practice, a short delay is configured into the polling mechanism loop. In other words, the master delays prior to polling the next node in order to reduce the network overhead required by the above ongoing protocol.

During normal operation, a slave node may also fail or leave the network. This is detected by the master through a poll response time-out. The associated slave-node ID is subsequently ignored in the polling cycle. Periodically, the master will re-poll all node IDs allowing nodes to enter or re-enter the network to achieve dynamic plug & play capability.

The “coup notice” message is associated with choosing or replacing a master node. Any node kernel may serve as the bus master. Naturally, when the network first boots up, there is no master. This situation also occurs if the current master is lost or removed from the network. Since polls are predictable and ongoing, the lost or absence of the bus master is easily detected by a slave node with a timeout. When a slave node detects the absence of a master as noted by an unexpectedly quiet bus, it attempts to assume the roll of master with two steps:

1. Idle for a period proportional to the node ID number.
2. Stage a coup by notifying all other nodes that it has assumed the master roll.

If a node receives a coup notice while idling (step 1), it abandons the attempt to become the master. It resumes the roll of slave and must re-register its sensors. The above algorithm results in the node with the lowest ID number becoming the new bus master. Our experience shows that this approach requires that the idle period must be at least twice the uncertainty associated with the time required for all nodes to detect the loss of the bus master.

The purpose of the last three “data dump” and “command” messages will be explained below in the section on satellite communications. The physical and network layer code is communications-interrupt driven. In other words, this software is invoked upon reception of incoming bytes from the communications bus.

3.2.3 Interface Layer.

On the applications side, the interface consists of the user-callable functions described above in section 3.1. Normally, an interface function is blocking does not return until the desired communication has been completed. Blocking also limits the amount of buffering space required within the kernel. The interface layer utilises the same thread as the applications code. Communications between the interface layer and the network/physical layers is via a CSP local channel.

The operation of the interface layer is best illustrated with a symbolic example showing the GetStatus() function call (fig. 4). In this diagram, *Send* and *Receive* represent intra-node or local channel communications while *NetTransmit* and *NetReceive* represent network communications.

Suppose the application on a particular slave node calls GetStatus() to retrieve status information on the global bulletin board for a particular sensor. The interface layer formats an appropriate data message containing this information and sends it the network/physical layer via a local channel. When a reply is received, the reply information is send back along the same channel.

When a network poll arrives at the network layer interrupt service routine for this node, it transmits the queued data message containing the GetStatus() information to the master node kernel and awaits a data message response. When the response data message arrives, the network layer routine a) transmits an end-of-file message to the master and b) returns a channel message to the blocked GetStatus() routine in the interface layer.

3.2.4 Program Control.

This unit resides in a reserved section of address space in the kernel. System messages for program control consist of reset requests or software uploads from the satellite. When a software upload message arrives, normal operation of application software is suspended and new code is written to program address space. Reset requests cause the program control unit to begin execution of application code at a pre-defined starting address.

```
// Interface layer routine called by
//    application thread.
int GetStatus(string sensor_name, int&
                    sensor_status)
{   format an appropriate network message
    Send request message on local channel;
    Receive reply message on local channel;
    Copy reply data to parameter
    return;
}
```

```
// Network/Physical layer
//        – invoked on poll for this ID –
void OnPoll() interrupt
{   if (local channel has a queued message)
      {  Receive request message from interface layer
         format network message
         NetTransmit message in response to poll;
         if (reply is appropriate)
         {   NetReceive reply from master;
             format reply data
            Send appropriate reply to application layer
         }
      }
   else NetTransmit an end-of-file reply
}
```

Figure 4. Symbolic code of interface and network/physical layer interactions for a PostStatus() call by an application

3.2.5 Bulletin Board Memory

The bulletin board represents a shared resource to the application programs on the network. The information posted by PostStatus(), PostSituation(), and PostData() calls resides centrally on the master node to simplify and reduce message traffic. The information posted by TelemetryData() calls resides locally upon each node.

The kernel does not backup or replicate the bulletin board, since this information is expected to be fairly transient and easily updated during normal operation. When a slave kernel determines that a coup has been successful, it must re-register its sensors.

3.3 Satellite Communications

The last three network messages defined in table 1 are utilised for communications with the satellite control computer and ground telemetry system. These system messages are transparent to the application programs.

As previously discussed, the satellite control computer views the sensor and experiment network as a single system. To communicate with the kernel, the satellite control a) issues a RTS discrete, b) pauses briefly, c) transmits a message, and d) awaits the reply. The pause allows the master node kernel to complete a polling cycle and be in an idle state.

To retrieve telemetry data to be sent to the ground, the satellite control transmits a “dump data” message. This message is received by the master node kernel. The master then sequentially issues specific “dump data” messages to each active slave in the network.

Upon receipt of a “dump data” message to a specific slave kernel, that kernel transmits the current data previously queued with a local TelemetryData() call. The satellite control computer simply receives these messages as if they were all being sent by a single node. Upon receipt of an end-of-file message with a node ID of zero, the satellite control computer releases RTS and allows the master node to proceed with network polling.

Uploading new executable code to sensor or experiment nodes from the satellite control computer is accomplished with a similar procedure using the “command” message.

4. Discussion

There are several costs associated with this self-configuring networking approach: In this first version utilizing the RS422 bus, the physical layer must monitor the bus for a poll and thus must be interrupt driven. In the current design, the physical layer may be configured to either perform as the system interrupt handler or be called by an operating system or scheduler. In the first case, the physical layer may invoke an operating system or scheduler interrupt handler if an interrupt is not communication related. In the second case, the physical layer assumes that all invocations are communications related.

A slave node kernel must monitor all bus bytes for a “poll” message. A processor can usually handle an incoming byte interrupt within a few instructions. Testing shows that even for a simple 12 MHz Intel 80C31 8-bit processor receiving at 19200 baud, the overhead of bus monitoring for “poll” messages represents only 2% of available cycles.

In this design, bandwidth does not scale: The bandwidth available to a node cannot exceed the bus rate divided by the number of nodes. Further versions of this system which replace the physical layer with higher speed intelligent interfacing may reduce overhead and increase transmission bandwidth. While this design can be timely, it may not be strictly real-time. In other words, the latency and lag of communications may not be strictly predictable.

The bulletin board situation integer is a shared global read/write resource. Updates to this integer are not atomic. As a result, it would be conceivably possible to loose an update according to transaction theory. (Sensor status and data does not have this problem.) This is not considered an issue however, since sensor situation changes should not be dependent on the previous state. In other words, it should not occur that a sensor would need to read the current situation in order to decide whether to post a situation change. Each sensor is expected to update specific situation bits based on measurements from nature.

While there are costs and issues to this approach, there are also several advantages to the self-configuring bulletin-board approach: The network kernel allows true *plug & play* and is tolerant of a variety of failures. There is no pre-designated bus master and the network is extensible. If a master node fails however, some current bulletin board data may be lost.

To facilitate parallel software development, a separate project will produce a simulator for the above network kernel. This simulator will run on a PC and interface to a single node via a PC serial port and a commercial RS232-to-RS422 adapter. The PC acts as the master kernel node. This will allow a programmer to test applications code with appropriate interface calls and actions. While the project is still in progress, preliminary results indicate the kernel will require approximately 8k of program space on each node.

There are a number of issues that will be considered in the future: The current design utilizes a scheduler to provide multiple threads and channel communications. If a simple node is responsible for a single sensor or experiment and does not require multi-tasking, it would be a simple matter to modify the kernel to support stand-alone operation. The physical and network layers would execute under the interrupt thread while interface kernel layer would run under the single application thread. Channel communication and blocking could be achieved with shared memory and a simple variable flag.

The use of the bulletin board model of communications may lend itself to full state fault-tolerance that insures no loss of data or state, as has been demonstrated in the MOM flight operating system [5-7]. In this similar system, nodes may arbitrarily be powered down and restarted for power budget management without any loss of state, processing, or data.

A more fault-tolerant design might replicate the bulletin board on each node. Certainly, the memory space required is expected to be minimal. Since each bulletin board update is broadcast on the bus, this would not require additional messaging. It would however, require additional processing on each slave node to keep the current copy of the board updated according to bus message traffic.

While this first kernel utilizes the RS422 serial bus, the design inherently supports any bus or open-link communications. This kernel could directly support satellite *clusters* using omni-directional RF or LED communications. All satellites in the cluster could be situationally aware and take advantage of all sensor capabilities. New satellites could be launched to join the cluster and older satellites could be de-commissioned without modifications to software or communications. Future work will examine these features and possibilities for the distributed kernel system.

In conclusion, we expect the completion of this project will allow significant time and budget reductions in the development of future small low-power satellites.

References

[1] Goforth T, Cannon S, Lyke J. "Space Qualification of the Advanced Instrument Controller", AIAA Conf. on Small Satellites, Aug 1999.

[2] Cannon S, Lyke J, Staggs J, Watson D, Fuller D. "A Parallel C30 Architecture for Miniaturized 3-D Monolithic Packaging", in Transputer Research and Applications, ed. Hamid Arabnia, IOS Press, pp 186-199. 1995.

[3] S. Ahuja, N. Cariero, and D. Gelernter, “Linda and Friends”, IEEE Comput. 19(8), 26-34 (1986).

[4] N. Carriero, D. Gelernter, “Applications Experience with Linda”, SIGPLAN Notices 23(9) 173-187 (1988).

[5] Cannon S and Dunn D. "Adding Fault-tolerant Transaction Processing to Linda." J. SOFTWARE -- Practice and Experience 24(5):449-466, 1994.

[6] Cannon S, Brinkerhoff D. “A Stable Distributed Tuple Space”, 29th Hawaii Internaltional Conference on System Sciences, Maui Hawaii, Jan, 1996.

[7] Hansen R, Cannon S, “An Efficient Fault-tolerant Tuple Space”, in Fault-tolerant Parallel and Distributed Systems, ed D. Avresky, IEEE Press, 1996.