The Virtual Mission Bus

Theodore M. Wong[1], Richard A. Golding[1], Harvey M. Ruback[2], Wilfred Plouffe[1], and Scott A. Brandt[1,3]

[1] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
[2] IBM Boca Raton, 8051 Congress Avenue, Boca Raton, FL 33487
[3] Computer Science Dept., University of California, Santa Cruz

Abstract. Distributed, adaptive, hard real-time applications, such as process control or guidance systems, have requirements that go beyond those of traditional real-time systems: accommodation of a dynamic set of applications, autonomous adaptation as application requirements and system resources change, and security between applications from different organization. Developers need a middleware with features that support developing and running these applications, especially as commercial and defense systems become more "network-centric". The *Virtual Mission Bus* (VMB) middleware, targeted at both distributed IT systems and real-time systems, provides the essential basic services to support these applications and the tools for building more complex services, all while while keeping the middleware kernel minimal enough for embedded system use. We successfully used the VMB to prototype a distributed spacecraft cluster system.

1 Introduction

Networked or distributed embedded systems are increasingly common and important. Some examples are new "network-centric" defense systems that integrate services in different locations and heterogeneous systems from multiple suppliers. Others include building, industrial, and energy-distribution systems that employ sensor and control networks. The flexible, distributed approach is also seen as part of the solution for avoiding cost and schedule overruns in government systems procurement [1].

We are developing the *Virtual Mission Bus* (VMB) middleware to provide features that support these applications. While these have the requirements of other real-time applications, they also require the flexibility to integrate multiple applications, some of which change after the system has been deployed, and the adaptability to respond autonomously to faults and application requirement changes. This leads to an integration of traditional middleware features and real-time resource management.

As an example, consider a simple control and data acquisition application that illustrates the integration challenges for flexible, real-time applications. It might consist of multiple basic components, each implemented as an independent process running on a node in a network: an interface to observe and command the system, sensors and actuators, a fault-tolerant real-time controller that periodically communicates with the sensors and actuators, and a data store to keep a record of system state. The sensors and actuators would need standard interfaces to isolate the controller from changes in the underlying hardware. The controller and data store would need to be fault-tolerant,

but with different approaches: the controller might use a hot standby, while the store might might use replication. On failure of a component, the application would need to recover using redundant components, and then try to find unused system resources to regenerate the failed component. The application should also try to maintain balanced resource usage by migrating components from heavily-loaded to lightly-loaded nodes.

Supporting flexible and adaptable real-time applications requires more than just the features of both existing middleware and existing real-time environments (§3). Most middleware is aimed more at IT applications than embedded systems, and focuses on communication between applications or on connecting application services; some provide a security model for these interactions. Most real-time environments provide for timely task execution or (soft) real-time data delivery between processes.

The VMB provides the features required by these applications with an architecture of simple, composable mechanisms implemented as libraries and reusable system components. Thus, simple embedded applications, such as sensor applications, need only a minimal number of mechanisms with a small runtime footprint and require the understanding of only a few system features. More complex applications can incorporate greater function by building on the basic mechanisms. The building-block approach extends to structuring the applications as components running on individual nodes.

The VMB provides real-time *resource management mechanisms* for CPU, memory, and network communication. The mechanisms ensure that application components have the required resources to execute on their assigned nodes, and *isolate* components so that they cannot interfere with each other. An application specifies the required resources for each of its component (§5.1) and uses a resource configuration mechanism to assign its components to nodes (§5.3). A resource scheduler ensures that each component gets its share (§5.2).

VMB application components interact with each other via an *RPC-like communciation protocol* (§6.1). Like other RPC protocols, components declare supported interfaces (§6.2). Unlike other protocols, ours is designed to be the building block for higher-level application-specific consistency and consensus mechanisms, by carrying message ordering information but not in itself maintaining consistency or message ordering.

The VMB includes experimental support for building reusable *application design patterns* (§7). These patterns capture common internal application structures—as a set of replicas, a primary-backup pair, a pipeline of data processing stages, and so on, or even combinations of these. A basic group membership mechanism, implemented over the VMB communication protocol, is the foundation for the reusable patterns.

Finally, the VMB supports application *security* (§8). These mechanisms serve both as internal defenses within an application, helping to contain faults from spreading, and as external defenses between applications, allowing only authorized accesses. The security features build on low-level resource isolation to protect individual components.

We have implemented a VMB middleware prototype (§9) and used it in the *Pleiades* system of interoperating clustered spacecraft under the DARPA System F6 program [2]. Pleiades has hard real-time guidance and control applications interacting with non-real-time data analysis applications, and requires fault tolerance and security for these applications. The prototype validates that the VMB fulfills requirements for distributed and real-time operation as well as requirements for flexible and adaptive systems.

2 Related work

The research on which the VMB is based is long and varied. We give references to some representative related work.

Middleware systems We can categorize the real-time, distributed systems middleware literature by the approach to communication. These approaches include RPC-oriented, message queueing, and group communication and consensus systems.

RPC-oriented systems build on RPC as a structuring communication primitive. The CORBA standard [3], Java RMI [4], and SOAP [5] are perhaps the most influential, and focus on request-response communication between a client and server, often in the context of an "object" on the server. The VMB shares the RPC-like pattern, but differs by focusing on RPC as a building block rather than a complete mechanism in itself.

Extensions to RPC-oriented systems provide real-time support. For example, the MEAD system [6] implements the real-time CORBA interface [7], and is one of several CORBA implementations that use a group communication system to support one-to-many communications to replicated services. MEAD adds adaptive replication mechanisms for fault tolerance to the real-time CORBA design, along with resource monitoring and proactive fault prediction and recovery. The VMB focuses on lower-level mechanisms than MEAD; one can build the replication mechanisms and proactive fault handling from MEAD atop VMB primitives. At the same time, the VMB provides greater support for heterogeneous systems where replication is not the appropriate fault tolerance strategy and where applications are explicitly assembled from reusable agents.

Message queuing and publish-subscribe systems, in abstract, have a global data space into which processes put and get data. The OMG Data Distribution Service [8] is one example among many. Data is often organized into topics or channels to which processes subscribe to receive asynchronous updates when data changes. Real-time versions have been developed that include deadlines, reliability, and data retention aspects. These systems are often implemented on top of lower-level communication systems. Publish-subscribe systems promote interoperable, flexible communication between system components, but they do not by themselves address many lower-level issues such as functional fault tolerance, functional consistency, or system resource management.

Group communication and consensus systems provide one-to-many messaging inside a defined group of processes. Some systems, such as the Horus and Ensemble [9] systems, yield a consistent (for example, totally ordered) and reliable view of message traffic. Other systems are built around consensus protocols, especially the Paxos [10] family of protocols, organizing process activity around a sequence of consensus decisions. These systems can be extended to support Byzantine fault tolerance [11]. The VMB communication protocol provides primitives that can implement group communication and consensus mechansisms, but that avoids the weight of these mechanisms when they are not appropriate. The VMB protocol supports building the consistency of sending *different* messages to a set of receivers, while group communication protocols typically send the same message to all receivers. This flexibility is useful when composing an application from multiple, pre-existing components that provide different operations that should nonetheless be performed consistently. The VMB does not currently attempt to address Byzantine fault tolerance, however.

Other distributed communication design patterns exist for organizing middleware-like systems. For example, peer-to-peer toolkits such as Chord [12] organize application processes by using an overlay network for routing data or queries; application reconfiguration is then cast as a overlay reconfiguration. The VMB provides components suitable for building peer-to-peer systems. Indeed, in an early prototype, we built a peer-to-peer publish-subscribe system using the basic VMB communication protocol.

Some middleware systems focus on sharing data, promoting interoperability between services by sharing database tables or distributed data structures such as a distributed hash table [13] or B-tree [14]. The VMB constructs enable application developers to implement shared data stores in a variety of ways. In an early prototype, we built a reusable VMB data store component from an embedded $\{key, value\}$ database.

Resource configuration and real-time The VMB manages resources using real-time schedulers for processes running on CPUs on individual nodes and for network traffic between nodes. Real-time CPU and network scheduling are well-investigated topics that have been successfully reduced to practice in many systems, but that are typically designed for one type of real-time processing.

Many traditional real-time CPU scheduling algorithms are based on either rate monotonic (based on static process priorities) or earliest deadline first scheduling (based on dynamic process priorities [15]). Both are designed for relatively static sets of hard real-time tasks. Neither provides for the dynamic, adaptable nature of VMB task sets.

Some more recent systems support adaptive real-time processes. The QRAM system [16] provides a solution for optimizing the resource allocations to sets of processes with (possibly adaptive) convex benefit functions relating their "quality" to the resources assigned to them. The DQM system [17] supports dynamic adaptive soft real-time applications with discrete quality of service levels. Neither supports a full range of different types of processes with the degree of adaptivity required for the VMB.

Other systems examine simultaneously management of processes with different types of timeliness requirements. The HLS Hierarchical Scheduler [18] uses multiple different schedulers in a hierarchy, where one scheduler would schedule time slots allocated to it from another. HLS demonstrated the feasibility of integrated scheduling, but proved too unwieldy for actual use.

The Constant Bandwidth Server [19] is one of the first scheduling frameworks to handle different types of real-time processes in a single scheduler. The RBED CPU scheduler [20] similarly supports multiple different types of real-time processes, hard and soft real-time, both adaptive and non-adaptive. The RBED CPU scheduler is based on the resource allocation and dispatching (RAD) model, which separately manages the amount of resources allocated to each process and the times of delivery of those resources. The VMB schedulers borrow concepts from the RAD model.

VMB resource configuration builds on previous research on the *replica placement problem* [21] of placing replicated data over a network to maximize client performance. Minerva system [22] and Ergastulum [23] investigated how to provision storage systems for a wide variety of workloads. Similarly, Harmony [24] provides a solution for placing storage and computation within a network to maximize overall performance. The VMB uses similar techniques to assign application components to nodes in a network.

3 Architectural requirements

The Virtual Mission Bus is middleware for distributed, adaptive systems that host multiple applications with heterogeneous real-time requirements. Thus, it must provide both the usual features for distributed systems, such as a communication model, interoperability mechanisms, and naming, and those for real-time systems, such as predictable, timely task execution and communication. Unlike many other middleware systems, it must also support applications for resource-constrained embedded environments.

Our target systems in general will host multiple interacting applications that are deployed at different times. For these applications to operate correctly, they must meet all their real-time behavior requirements (even as other applications change around them) and communicate properly with new applications. Most existing real-time systems do not protect applications from the behavior of each other, so their development processes emphasizes exhaustive pre-deployment testing of static configurations and disallows post-deployment change. In contrast, the VMB *isolates* applications from each other.

Different applications will have different real-time requirements. Control systems are usually hard real-time, while data analysis can be soft real-time. Other applications may not have specific real-time requirements, but they must coexist with applications that do. Keeping in mind that applications can be added or removed dynamically, this means the VMB needs to support a model of *resource management* that can correctly intermix these heterogeneous requirements in one system.

Application components will need to communicate using an appropriate communication model. Nearly all middleware systems provide a communication mechanism, but most are designed around a single model, thus imposing structure on applications; for example, data distribution services structure applications on data subscriptions, and group communication services structure applications as process groups. The VMB takes a building block approach, via *reusable components* and *application design patterns*, that facilitates implementation of and reasoning about several communication models.

Many applications require autonomous fault recovery and adaptation. Applications often run redundant components in different fault domains (for example, control systems often use a primary-backup structure, while data stores use n-replication). When a fault happens, an application must recover from the fault by reconfiguring itself (for example, by activating a backup), and should try to restore redundancy by obtaining resources and starting a replacement component. Similarly, when application requirements or available system resources change, an application must be able to migrate or reorganize its components accordingly. The VMB provides *resource configuration* mechanisms for an application to specify resource requirements and allocate resources.

Finally, many of our target systems will host applications from different organizations, and with different classification levels. Some of the applications are critical to the safe operation of expensive systems and vital national assets. Thus, applications must be protected from each other, from attacks from the outside, and from faults on the inside of an application. The requirements for security complement the requirement that real-time application components must be isolated from each other. The VMB *security* mechanisms allow a designer to reason about how heterogeneous systems will interact, which is an essential building block for certifying systems with dynamic application populations.

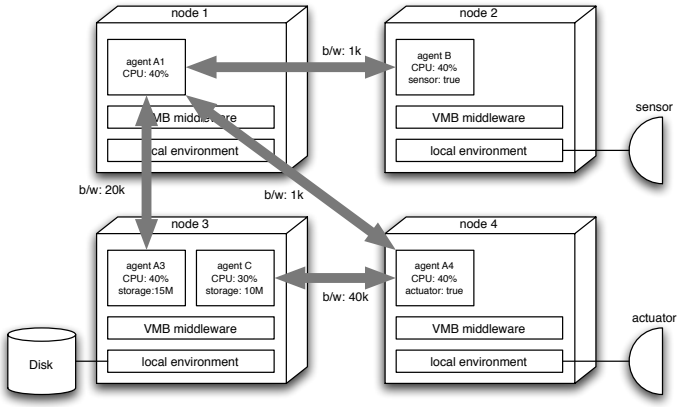

Fig. 1. VMB architecture. The example shows a system of three applications running on four nodes. Each application consists of one or more agents that communicate with each other through microtransactions. Different nodes provide different specialized hardware, such as storage, sensors, or actuators.

4 Virtual Mission Bus design

In the Virtual Mission Bus, an *agent* is the executable building block of a system. An agent is code that executes on one node and consuming local computing, memory, and storage resources. Nodes can host multiple agents. An application is then a collection of one or more agents that together perform some useful function. Each agent in an application can have a different role, such as managing the application, performing control decisions, storing sensor data, and so on. In the example in Figure 1, Application A consists of three agents, running on nodes 1, 3, and 4.

The VMB is implemented as middleware running on each node. Depending on the local operating system, part of the VMB executes as part of an agent process, while other parts execute as service processes or in the OS kernel, as needed.

The VMB manages the resources on a node (§5). Each agent can request a certain resource allocation, and the VMB works with the local OS to enforce that allocation. The corollary is that each agent is isolated from other agents: an agent will always get its allocation, and will not overrun its allocation unless there are unallocated resources on the node (modulo security issues (§8)). The application is responsible for requesting sufficient resources for its agents to complete their work. In the example, each agent has reserved some of the CPU on its node.

Agents communicate through a *microtransaction* protocol that provides RPC-like communication (§6). The protocol supports a simple one-phase read version where operations can take effect immediately at receivers, and a two-phase update version where operations do not take effect unless all of the receivers have agreed to the operations. As in shown Figure 2, the protocol supports both one-to-one and one-to-many operations. The protocol adds ordering information to operation messages so that applications can build consistency and ordering mechanisms that provide an appropriate degree of

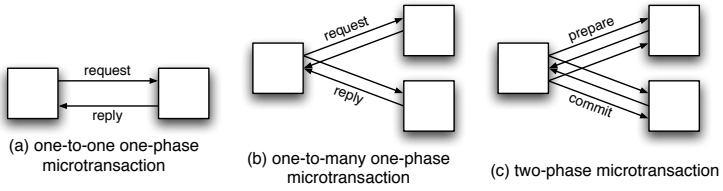

Fig. 2. One- and two-phase microtransaction operations.

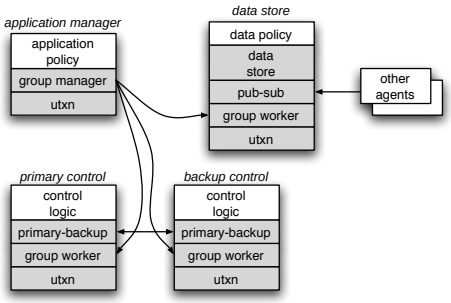

Fig. 3. Reusable design patterns in the VMB. Grey boxes are reused *code packages* or libraries; they encode common patterns such as a manager-worker pattern for group membership and a primary-backup pattern for failover.

atomicity or serializability. In the example in Figure 1, the agents from Application A communicate with each other, and with agents from Applications B and C.

Many communications between agents have real-time constraints, and so the VMB supports real-time network communication. An application can reserve network performance (§5) between pairs of agents. Each message that is sent is then associated with one reservation, and its transit through the network is scheduled accordingly.

The scheduling and communication primitives make up the core of the VMB, on which we build reusable patterns for more sophisticated features, as shown in Figure 3. The VMB provides three features for building these patterns: *code packages* that package up a defined piece of functionality to be used in agents (typically in a shared library, depending on the OS), *reusable agents* that provide common application functions such as data stores or fault tolerance monitors, and *application structures* that capture encode agent arrangements, such as primary-backup, replication, or data pipelining.

Since all applications must share system resources, the VMB supports coordinating resource allocation (§5.3). In our past work [25] we investigated algorithms to allocate resources across multiple applications based on their local requirements. In the VMB we extend this work to consider network communication requirements. In the normal case, each application computes its own optimal placement, based on the resources available from nodes it trusts. But in some cases, multiple applications must reconfigure themselves together, and so defer to a shared resource management service. In the example in Figure 1, Application A originally queried all four nodes in the system, com-

puted an appropriate resource allocation, and created its agents. If the application later needs to (say) increase the amount of computation done by its agent on node 3, it will determine how much CPU resource is available on different nodes, and may increase the allocation for the existing agent or migrate the agent to a different node.

The VMB supports security where required by applications (§8). The building blocks are a certificate-based mechanism for identifying and authenticating agents and applications to each other, a method for encoding access rights and delegating them from one principal to another, and encryption negotiation based on the certificates, along with the resource isolation provided within a node. Security relationships such as trust among the members of an application can be encoded using these primitives. In the example Figure 1, each agents in Application A proves its membership with a certificate; the application overall identifies its owner with another certificate. When the agent on node 1 sends an operation to the agent from Application B on node 2, the receiver verifies the certificate chain of the sender, and checks that Application A has a credential issued by the owner of Application B that granted permission for the operation.

Finally, the VMB provides some basic distributed services. The two most important are *naming* and *code distribution*. The name service provides a graph-structured name space for the system, modeled after that of the Amoeba distributed operating system [26]. The code distribution service provides a secure mechanism for transporting agent implementations between nodes. Naturally, we build both of these services from core VMB primitives.

5 Resource management and scheduling

The Virtual Mission Bus supports real-time control systems, and so it needs to control resource usage precisely. This includes CPU time, memory, and network communication; a fault-tolerant or secure applicatiosn need stable storage as well.

The VMB *isolates* agents from interfering with each other. Each agent is given an allocation of CPU, memory, storage, and communication resources, and each resource is managed so that the agent will get its allocation, no matter how other agents behave.

There are two reasons we focus on isolation: system correctness and security. It must be possible to add a new application to a running system, validate that its resource requirements are met, and be sure that both the new application will execute correctly and that the new application cannot disturb the correct execution of existing applications. Compare this to standard practice in most embedded system development, where all of the tasks that can be performed in a system must be defined, validated, and tested a priori, before deployment, and at runtime lack internal defensibility against system components that misbehave.

Our approach is to formalize resource specification, and then divide resource management into two layers: a long-term resource configuration problem, which includes the traditional task of admission control, and short-term resource scheduling.

5.1 Resource specification

The VMB supports a wide range of kinds of agents and applications, from synchronous embedded controllers through to interactive applications at the edges of systems. Some

of these require hard real-time task execution, while others do not. Some use significant amounts of CPU time, while others consume storage and network bandwidth. Different applications take different approaches to fault tolerance. Table 1 summarizes the resources that the VMB manages, and how their requirements are specified.

Table 1. VMB resource specifications

Resource	Units
Agent CPU time	baseline CPU seconds per period
Agent memory	bytes
Agent stable storage	bytes
Agent-to-agent network	bytes per period
Agent placement constraints	same or separate node or cluster as other agent(s)
	maximum network latency to other agent(s)

The VMB uses the Resource Allocating/Dispatching (RAD) model [20] for resource specification. This model, original developed for CPU scheduling and since extended for disk [27] and network scheduling, supports hard and soft real-time processes in a single framework by specifying the amount of the resource that is needed and the period in which the resource is needed. The admission control algorithms for this model are simple under reasonable assumptions of task granularity.

Since each node may have a different model processor, the amount of memory and CPU cycles to achieve a particular task may vary from node to node. We are currently using a simple approach for allocation, in which an application specifies that it needs x CPU seconds per period on a baseline processor, and each node is treated as providing the equivalent of y of these baseline CPU seconds per second. This approach suffices for the embedded processors we have today. In the future, we plan to investigate more sophisticated portable representations.

Network bandwidth is also specified using a RAD model, by the amount of bandwidth between two agents and the period on which the bandwidth should be available.

The VMB resource specification includes constraints for ensuring that agents are on different nodes or groups of nodes, and that network communication latency is within some bound. A fault tolerant application might require that redundant agents are in separate failure domains.

Some agents require minimal resource control. Consider a web server that is the interface between a sensor system and the larger Internet: it needs interoperability with the real-time systems but is not itself real-time. This kind of system can specify that it only needs best-effort resource management.

5.2 Short-term resource scheduling

When an agent is created, it is assigned a certain amount of node and network resources. The resource configuration and admission control systems—discussed next—ensure that this assignment is feasible, so once the resources are assigned it is up to the nodes and networks to ensure that the agent gets its those resources. This calls for scheduling.

For memory and stable storage, each agent is given an amount and must work inside that amount. Operating system mechanisms are used to enforce these limits.

CPU resources are controlled by the operating system's scheduler. For real-time systems, we replace the existing CPU scheduler with the RBED scheduler [20], which uses the RAD scheduling model.

The VMB also provides a high-level API for managing real-time tasks. This API allows an agent to declare explicit tasks that need to be performed, the time at which those tasks should be started, and get feedback on whether the tasks are completing on time. The tasks execute in the context of the agent's CPU reservation, and so the agent must ensure that the reservation matches the tasks that will be executed.

We are investigating network resource management, adapting the work that has been done to date on real-time network protocols, and on scheduling in IPv6 network stacks, to use RAD-style scheduling. Our current design uses the flow label field in IPv6 packets to associate packets with a network reservation, and has packet schedulers in senders and routers to ensure that each reservation is honored.

5.3 Long-term resource configuration

Each application in a system needs to autonomously allocate enough resources for each of its agents, even as the system changes. This includes the traditional admission control problem; once a feasible allocation is made, short-term scheduling will ensure correct execution. Long-term configuration decisions are made when node or network resources change and when application requirements change.

A good resource allocation will accommodate as many applications as possible while respecting fault independence and latency constraints. In many cases this involved keeping communicating agents as close together as possible to minimize the total network resource they use.

We have developed a family of heuristics for this kind of problem, based on heuristics for the multidimensional bin packing and knapsack problems. These derive from our earlier work [28, 25], where we demonstrated heuristics that configured agents without considering network resources. These heuristics build on the Toyoda heuristic [29] and the variant we used in the Minerva system [22]. Our new heuristics add network resources to the n-dimensional resource vector space considered when computing how well an agent fits onto a node.

Our approach combines centralized and decentralized solutions. Centralized decision-making is easier to implement and can yield better results because it has a global view of system state. However, "centralization" is not easily defined in a system that is potentially very large in scope, is not necessarily fully connected (especially in fault situations), and where security considerations prohibit sharing configuration information. We decentralize the configuration problem as much as possible, where each application includes one *manager* agent (§7) that is responsible for computing an allocation for the application's agents based on the resources it already has and the resources that trusted nodes report they have available. In some cases, however, configuration needs to be centralized—in particular, when recovering from a node failure. This usually occurs when some application that was not directly affected by the failure must be reconfigured to make room for another application (for example, so that the affected application can

keep all its agents on different nodes for fault tolerance). The centralized configuration heuristic attempts to find a small set of applications to jointly reconfigure.

6 Communication between agents

6.1 Microtransaction protocol

All VMB agents communicate via a microtransaction protocol that provides RPC-like request-response operations. As shown before in Figure 2, the protocol supports a one-phase read version and a two-phase update version. A sender can group one or more one-phase or two-phase operations together into one microtransaction, provided that all of the operations are only of one kind or the other. The protocol is based on our earlier work on simple protocols for consistent operations in distributed systems [30, 31].

Microtransactions provide the building blocks for consistent ordering within the system, assuming crash failures in a timed-asynchronous system model. The protocol uses orderings based on timestamps, as opposed to orderings based on locking or on a centralized ordering service. A sending agent tags its outgoing microtransactions with a globally-unique timestamp. A receiving agent tags a data object that microtransactions can manipulate with two timestamps: the most recent read of the data, and the most recent update to the data. An agent, broadly speaking, will reject a microtransaction with an earlier timestamp than the ones on the data. An agent can generate timestamps from any monotonically increasing clock that is loosely synchronized to clocks at other agents and has fine enough granularity to produce unique timestamps, but an agent with a slow clock may find that its microtransactions never succeed.

The protocol does not by itself enforce consistency and durability properties. The receiver is responsible for performing application-specific integrity checking on incoming microtransaction operations, and for logging updates on data objects to stable storage.

6.2 Interfaces

Each VMB agent exports one or more interfaces that define the operations it can process. The interfaces are expressed in a simple IDL. Each interface has a globally-unique name, following the naming convention used for Java packages of the DNS name of the sponsoring organization with elements in reverse order, followed by a path name for the specific interface—for example, `com.ibm.vmb.example.interface`. Interfaces are also have version numbers. An interface imposes a syntactic, but not semantic, constraint on an agent.

Each operation in the interface is defined as the operation "type" (one-phase or two-phase), a name for the operation, and a signature of typed input and output parameters. The parameter types include basic numbers and strings, references to application-specific types, and variable-length arrays of these types.

A VMB interface represents an informal contract between sender and receiver. The sender can expect that the receiver supports all of the operations named in an exported interface, and that the functionality of the underlying implementations conforms to some externally-defined standard. The version number on an interface helps to identify when the expected functionality changes, even if the operation signatures do not

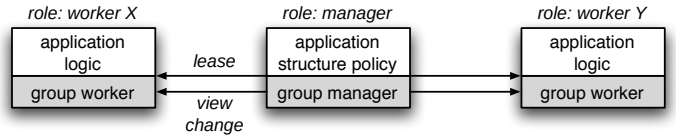

Fig. 4. Group services design. A manager runs a failure suspector and coordinates membership.

change. In addition, one interface can include another: if an agent exports interface A, and interface A includes interface B, then the agent must also support interface B. This allows for creating interfaces that represent a profile of multiple interfaces that a particular kind of agent is expected to support.

The VMB includes IDL processor tools that generate sender and receiver stubs. The code for each interface is generated separately, so that support for some interfaces can be packaged into reusable code packages, while other interfaces are supported directly by agent-specific code.

The interface system supports implementing higher-level communication patterns as libraries that can be reused in multiple applications. We have to date implemented group membership, totally-ordered group communication, and one-to-many publish-subscribe mechanisms (§7). The consistency mechanisms in the microtransaction protocol allow these communication patterns to work together in a consistent way.

7 Application design patterns

The VMB provides parameterized building blocks to support a range of distributed application design patterns. These building blocks, in the form of libraries that export microtransaction interfaces (§6) or local APIs, provide a variety of common application support services, such as protocols for managing consistent views of global state, mechanisms for scheduling and executing real-time tasks, or buffering data in a distributed processing pipeline. The goal is to strike a balance between APIs that provide arbitrary flexibility but no support, such as simple messaging, and those that provide support by forcing applications into one structure, such as group communication.

A key decision when implementing a distributed application is in assigning specialized roles to agents in an application. An application may organize its agents into primary-backup, N-replicated, or pipelined structures, depending its data flow and fault tolerance requirements. The VMB group services implements a basic manager-worker agent pattern to provides failure detection and global state maintenance mechanisms, as shown in Figure 4. The application, by using the manager to assign roles to workers, can then use the group services as the foundation for implementing higher-level patterns.

7.1 Group services

The VMB group services support the structuring of an application in a manager-worker pattern. The current group services include a *lease* protocol for mutual detection of agent failures, a *view-change* protocol for maintenance of consistent global application

state, an *election* protocol for selecting a new manager, and a *join* protocol for bringing new workers into an existing group. These protocols are implemented in worker and manager libraries that implement the appropriate interface, and call into application logic on key group events. The group services are based on the Palladio replica management protocol [32].

The lease protocol enables a manager to periodically test if a worker is still alive. To test a worker, a manager sends a one-phase lease request to a worker containing a new lease expiry time. If a worker does not receive a new lease from its manager before the current expiry time, it believes that the manager has failed. Likewise, if a manager does not receive a reply to a lease request before the current expiry time, it believes that the worker has failed.

The view-change protocol enables a manager to push changes in the global metadata view of an application to all workers consistently. Basic metadata includes a group membership list of the manager and all known live workers; an application may add other information such as application security credentials. To propose a new view, a manager starts a two-phase view-change operation with all known live workers. If the workers acknowledge receipt of the new view, the manager can commit to the view.

7.2 Implementing higher-level patterns using group services

The manager-worker pattern is a foundation for many higher-level patterns. The manager supports implementation of these patterns by assigning distinguished roles to its workers, and by using the view-change protocol in the group services to ensure that all workers have a consistent view of their roles. Some higher-level patterns include:

- Primary-backup servers. The manager designates one worker in a group as a primary server, and other workers as backup servers. If the manager detects the failure of the primary, it redesignates one of the backups as the primary.
- N-replicated servers. The manager designates N workers as replicas. If the manager detects the failure of a replica, it spawns a new replica and assists with synchronizing the state with that of the existing replicas.
- Pipelined processing agents. The manager associates each worker with a pipeline stage. Alternatively, a set of unmanaged agents can arrange themselves in a pipeline outside of the group services, but without the benefit of a manager to detect failures.

The evaluation CF and imaging applications (§9.2) uses the primary-backup and pipeline patterns.

7.3 Other common real-time application patterns

The VMB includes several other shared library APIs and reusable agents, including:

- A *timer services* API for scheduling and executing hard real-time tasks.
- Publish-subscribe style communications between data producers and consumers. A set of VMB publish-subscribe interfaces, built on microtransactions, enables consumers to receive data updates from producers, either on a regular schedule or as new data comes available.

– A generic *data store* agent. The VMB provides a data store agent that keeps *(key, value)* pairs in a database, and that developers can customize with a maximum capacity and set of record retention policies. The data store supports the publish interface for pushing records to a downstream consumer, and the subscribe interface for pulling records from an upstream producer.

8 Information assurance

The VMB integrates information assurance (IA), or security, into its basic structure. Some systems need security because they incorporate applications from multiple organizations or that run at different security levels. Other systems do not intrinsically have multiple security levels, but they share infrastructure—such as shared networks—with non-trusted applications.

8.1 Levels of protection

The VMB IA architecture is flexible because it has to accommodate applications and users that have widely different threat models to defend against.

The simplest systems have one level of IA and act on behalf of one organization. These systems do not need "security" in the traditional sense; however, our experience is that simple protection mechanisms are worth implementing because they help contain faults, rather than to protect against attack. For example, computing a simple message checksum or including an "authentication" handshake when establishing a communication session helps to detect application bugs.

A collection of commercial-grade applications sharing an untrusted system forms an intermediate IA level. These systems need communication encryption and authentication to protect against network-based attacks; network resource management to mitigate some forms of denial of service attacks; and authorization checks that users are not attempting to misuse the system. These attacks can be addressed using commercial-grade security practices.

DoD multilevel security (MLS) addresses the most complex threats, including covert channels for information disclosure. It requires system policies for mixing data from different sources, and rules for the classification level derived information products. These concerns require certified IA systems, including OS kernels and networks.

8.2 Design approach

We are currently designing and implementing the VMB information assurance mechanisms. They are modular, so that different applications can select mechanisms appropriate to their threat model. We are implementing commercial-grade solutions first, and will extend to MLS solutions in future.

The resource-isolated agent is the basis for security. Each node provides a trusted computing environment that ensures the integrity of the code that agents run, its memory, and its storage. Protected storage allows an agent to store identity and security credentials safely. This assurance depends critically on operating system protection mechanisms.

Agents that communicate using the microtransaction protocol use authenticated and encrypted communication sessions. At present we are using the Internet Transport Layer Security 1.2 standard [33], which uses X.509 certificates to distribute and authenticate public-key encryption pairs, and negotiates symmetric session encryption keys.

Each operation that one agent sends to another is checked for authorization. Authorization derives from *sponsoring principals* that delegate rights to users, who delegate in turn to applications, and from there to agents. These rights are encoded as metadata included in X.509 certificates. Sponsoring principals act as certificate authorities, and rights delegations are represented as links in a certificate chain.

Typical usage is for service provider and service consumer organizations to first establish a contractual trust relationship, represented as cross-certificates that encode the usage rights that the service provider is delegating to the consumer. The consumer organization delegates those rights to its applications; each application has a certificate representing the application as a whole. The application in turn delegates rights by issuing certificates to the agents that make up the application. Rights can be restricted at each step, so that (for example) only one agent ends up with the operational right to make a request of the service provider's application.

The services that all applications use, such as naming and resource management, are designed for security and learn from our experience with existing Internet services. Naming, for example, is implemented as a federation of name services, each of which is operated by a known service provider. The authentication and authorization mechanisms allow one to be sure that the name system is not being tampered with or spoofed. This approach draws on the efforts to improve the Internet Domain Name Service [34].

9 Implementation and evaluation

We are implementing the Virtual Mission Bus as part of the Pleiades clustered spacecraft system [2] for the DARPA System F6 program [35]. System F6 should demonstrate that a "fractionated" cluster of small, individually launched spacecraft connected by a wireless network can deliver better value and utility than an equivalent traditional, large, monolithic satellite. With Pleiades, our approach is to treat every hardware element—mission-specific payload components, spaceflight components, and ground terminals—as a VMB node in a distributed computing system, and transform the control software into a set of distributed applications. A unified IPv6 network built over the intra-spacecraft, inter-spacecraft, and space-ground wired and wireless links connects together all of these elements. VMB application agents manage every aspect of system operation, including mission sensing and actuation, independent spacecraft flight, cluster flight, and ground command and control.

In the first phase of the F6 program, we evaluated the VMB architecture for Pleiades by building a *hardware-in-the-loop* (HIL) cluster test bed, implementing prototype versions of much of the VMB and application software, and demonstrating key system operation tasks. Pleiades, even with only an unoptimized prototype of the VMB, is already able to accomplish its real-time command and control tasks.

9.1 Hardware-in-the-loop test bed

The hardware-in-the-loop (HIL) test bed is a simulation environment used to validate VMB functionality and other Pleiades components. The HIL test bed includes accurate physics models of satellite systems, and an accurate enough computing environment validate the feasibility of using the VMB to implement distributed spacecraft systems. The HIL test bed named for its supports of incremental, transparent insertion of actual flight hardware components in place of their corresponding simulated components.

The test bed consists of x86 Linux and Microsoft Windows™ server pairs that each simulates a Pleiades spacecraft, connected by Ethernet networks. Spacecraft physics, such as position, attitude, and sensor and actuator states, are simulated with modified version of the Dynamic Spacecraft Simulator (DSS) from Orbital Sciences Corporation running on the Windows server. Another x86 Linux server hosts all ground agents, including those that send commands to and retrieve telemetry from the spacecraft.

Virtual machines (VMs) hosted on the Linux server in a pair simulate the component nodes of a spacecraft, and a virtual IPv6 intra-spacecraft network connects the nodes. Component nodes include the flight control processor, flight sensors and actuators, and mission payload. A Wind River VxSim VM running the VxWorks 6.6 operating system implements each node, and the VxSimNetDS package implements the intra-spacecraft network. For the prototype, the memory of each VxSim is set to 128 MB.

A router component node in each spacecraft routes communications from the intra-spacecraft network to the inter-spacecraft or space-ground networks. All inter-spacecraft and space-ground communications pass through a network impairment driver. This driver allows the test bed to emulate the network latencies and conditions associated with space flight, such as the variability of bandwidth or error rates with distance, or the visibility of ground stations from a spacecraft.

9.2 Utility evaluation

The VMB middleware enables the rapid development of modularized real-time control applications, particularly in environments previously dominated by monolithic systems designs. We demonstrated this by successfully developing three interdependent applications in Pleiades for basic spacecraft flight control, cluster guidance, and mission payload operation. A different team led the development of each application, but our approach of using microtransactions coupled with interface definitions (§6) provided the clear module boundaries needed to integrate the applications into a unified system.

In the first evaluation step, we successfully validated the basic microtransaction and IDL design, by migrating existing monolithic spacecraft flight control software (FSW) and the ground control applications to more flexible VMB-based implementations. We modified a legacy FSW application to communicate with spacecraft flight sensor and actuator components through microtransactions over an intra-spacecraft IPv6 network rather than through local signaling channels (such as serial lines), but otherwise retained the original flight control logic; for each component, we defined a new microtransaction interface and transformed the component into a node with a subsystem agent that exported the interface. We then modified a legacy ground control application to communicate with the new VMB-based FSW instance on each spacecraft through a protocol

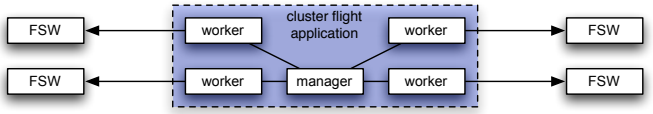

Fig. 5. Cluster flight software structure. A worker on each spacecraft in a cluster, in cooperation with the spacecraft FSW, makes local guidance decisions. A manager on one of the spacecraft in a cluster monitors workers, and nominates one worker to make cluster-wide guidance decisions.

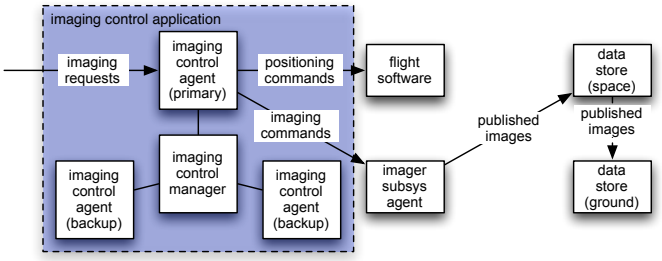

Fig. 6. Imaging application structure. An imaging control application, consisting of a primary and backups watched by a manager, directs independent subsystems to maneuver a spacecraft and capture images. A data store buffers images when waiting for a space-ground link to open.

translator. We confirmed through simulations on the HIL test bed that the VMB-based applications were operationally equivalent to their monolithic predecessors.

In the next step, we successfully designed and implemented a VMB-based distributed real-time control application, by designing and implementing the cluster flight software (CF) application (Figure 5). The CF employed the manager-worker pattern to structure its agents. A worker agent on each spacecraft in a cluster made local guidance decisions, relying on the spacecraft FSW instance for maneuvering. A manager agent on one of the spacecraft in a cluster detected failed workers, and nominated one worker to make cluster-wide guidance decisions. Each worker agent used the timer services API (§7.3) to managed its regular control cycle. We confirmed through simulations on the HIL that the CF correctly maintained spacecraft in relative cluster orbits. We also demonstrated the utility of the building-block paradigm, by having the CF worker agents developed by one team interoperate with the FSW developed by another team.

In the final evaluation step, we successfully demonstrated the utility of the VMB building-block paradigm by developing an Earth observation and imaging mission application (Figure 6) that used previously developed applications as services. An imaging controller agent used the timer services API to translate a request to capture an image of a set of coordinates into a schedule of spacecraft operations: slewing the spacecraft (via the FSW) to point an imager component at the coordinates, holding the spacecraft stationary after the slew (again via the FSW), and capturing an image; a manager agent enabled switching from an existing controller agent on one node to a new controller on

another node. The imager subsystem agent published captured images to a buffering data store (§7.3) in space, which in turn published the images to a data store on the ground when a space-ground link became available. Reuse of the FSW application, as well as built-in VMB support of common real-time application patterns (§7.3), enabled us to limit the imaging application development time to less than ten person-months.

9.3 Performance evaluation

Our prototype VMB system performs correct command and control operations for distributed real-time systems. We demonstrated this ability in a HIL simulation that consisted of seven spacecraft running the FSW, CF, and imaging applications. Our results showed correct simulated operations in real time and in "fast-time" at five times real time. The "fast time" result indicates that the virtual nodes were at most 20% utilized in real-time operation, as synchronization mechanisms in the DSS disallowed running faster than correct physics behavior would allow. As we focus in future phases on optimizations for speed, efficiency, and memory, we expect performance to improve further.

10 Conclusions

The Virtual Mission Bus takes the architectural approach of providing a minimal set of features that promote building consistent, reusable higher-level features for distributed, adaptive, real-time applications. This approach allows an application to use the structure that best fits it, and helps to avoid using scarce processor resources on unneeded features. The result is that the VMB low-level features are complementary to higher-level middleware services. We have implemented a straightforward group membership and communication mechanism, redundant application component policies, publish-subscribe communication, and reusable agent designs in a VMB prototype.

Our experience building and testing the Pleiades clustered spacecraft system validates our approach. Teams at three different organizations have used the VMB to implement their portions; the pieces integrated smoothly and properly controlled a simulated system of multiple spacecraft. The VMB design evolved while building the Pleiades system, most notably adding security and publish-subscribe features—using existing mechanisms as building blocks–as we came to understand the requirements for them.

We are continuing research on resource scheduling and configuration, the micro-transaction protocol, and security, and continuing development on the VMB prototype.

Acknowledgments

We thank the many people who have participated in VMB research and development, including: the VMB development team in the IBM Systems and Technology Group, the Pleiades development teams at Orbital Sciences Corporation and the Jet Propulsion Laboratory, our collaborators on the Collective Intelligent Bricks/K2 project, and our collaborators at UC Santa Cruz. We also thank Chris Hanson at IBM Research for helping to build the business case for the VMB. We gratefully acknowledge our funding from the DARPA System F6 program, under DARPA contract #HR0011-08-C-0031.

References

1. Brown, O., Eremenko, P.: The value proposition for fractionated space architectures. In: AIAA SPACE Conference and Exposition. (September 2006)
2. LoBosco, David M., Cameron, G.E., Golding, R.A., Wong, T.M.: The Pleiades fractionated space system architecture and the future of national security space. In: AIAA SPACE Conference and Exposition. (September 2008)
3. Object Management Group: Common Object Request Broker Architecture (CORBA/IIOP), version 3.1. (January 2008)
4. Sun Microsystems, Inc.: Java remote method invocation. `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/`, retrieved 1 April 2009
5. XML Protocol Working Group, World Wide Web Consortium (W3C): SOAP Version 1.2. Second edition edn. (April 2007)
6. Narasimhan, P., Dumitras, T.A., Paulos, A.M., Pertet, S.M., Reverte, C.F., Slember, J.G., Srivastava, D.: MEAD: support for real-time fault-tolerant CORBA. Concurrency and Computation: Practice and Experience **17**(12) (2005) 1527–45
7. Object Management Group: Real-time CORBA specification. (January 2005)
8. Object Management Group: Data Distribution Service for real-time systems, version 1.2. (January 2007)
9. Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., Vogels, W.: The Horus and Ensemble projects: Accomplishments and limitations. In: Proc. DARPA Information Survivability Conference & Exposition I. (January 2000)
10. Lamport, L.: The part-time parliament. ACM Trans. on Computer Systems **16**(2) (May 1998) 133–69
11. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proc. the 3rd Symp. on Operating Systems Design and Implementation, USENIX Assoc. and ACM SIGOPS (February 1999) 173–186
12. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for Internet applications. IEEE/ACM Trans. on Networking **11**(1) (February 2003) 17–32
13. Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M.F., Morris, R.: Designing a DHT for low latency and high throughput. In: Proc. First Symp. on Networked Sys. Design and Impl. (2004) 85–98
14. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L.: Boxwood: abstractions as the foundation for storage infrastructure. In: Proc. the 6th Symp. on Operating Systems Design and Implementation. (2004) 105–120
15. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the Association for Computing Machinery **20**(1) (January 1973) 46–61
16. Rajkumar, R., Lee, C., Lehoczky, J., Siewiorek, D.: A resource allocation model for QoS management. In: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997). (December 1997) 298–307
17. Brandt, S., Nutt, G., Berk, T., Mankovich, J.: A dynamic quality of service middleware agent for mediating application resource usage. In: Proceedings of the 19th IEEE Real-Time Systems Symposium. (December 1998) 307–317
18. Regehr, J., Stankovic, J.A.: HLS: A framework for composing soft real-time schedulers. In: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, IEEE (December 2001) 3–14
19. Abeni, L., Buttazzo, G.: Resource reservation in dynamic real-time systems. Real-Time Syst. **27**(2) (2004) 123–167

20. Brandt, S.A., Banachowski, S., Lin, C., Bisson, T.: Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In: Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003). (December 2003) 396–407
21. Szymaniak, M., Pierre, G., van Steen, M.: Latency-driven replica placement. In: Proc. Symp. on Appl. and the Internet (SAINT). (2005) 399–405
22. Alvarez, G.A., Borowsky, E., Go, S., Romer, T.H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., Veitch, A., Wilkes, J.: Minerva: An automated resource provisioning tool for large-scale storage systems. ACM Trans. on Comp. Sys. **19**(4) (November 2001) 483–518
23. Anderson, E., Kallahalla, M., Spence, S., Swaminathan, R., Wang, Q.: Quickly finding near-optimal storage system designs. ACM Trans. on Computer Systems (TOCS) **23**(4) (November 2005) 337–74
24. Singh, A., Korupolu, M., Mohapatra, D.: Server-storage virtualization: integration and load balancing in data centers. In: Proc. ACM/IEEE Conf. on Supercomputing. (2008)
25. Golding, R.A., Wong, T.M.: Walking toward moving goalposts: agile management for evolving systems. In: Proc. First Workshop on Hot Topics in Autonomic Computing (HOTAC-1). (Jun 2006)
26. Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G.J., Mullender, S.J., Jansen, J., van Rossum, G.: Experiences with the Amoeba distributed operating system. Comm. ACM **33**(12) (1990) 46–63
27. Povzner, A., Kaldewey, T., Brandt, S., Golding, R., Wong, T.M., Maltzahn, C.: Efficient guaranteed disk request scheduling with fahrrad. In: Eurosys 2008. (April 2008)
28. Wilcke, W.W., Garner, R.B., Fleiner, C., Freitas, R.F., Golding, R.A., Glider, J.S., Kenchammana-Hosekote, D.R., Hafner, J.L., Mohiuddin, K.M., Rao, K., Becker-Szendy, R.A., Wong, T.M., Zaki, O.A., Hernandez, M., Fernandez, K.R., Huels, H., Lenk, H., Smolin, K., Ries, M., Goettert, C., Picunko, T., Rubin, B.J., Kahn, H., Loo, T.: IBM Intelligent Bricks project—petabytes and beyond. IBM J. Res. and Dev. **50**(2/3) (March/May 2006) 181–197
29. Toyoda, Y.: A simplified algorithm for obtaining approximate solutions to zero-one programming problems. Management Science **21**(12) (August 1975) 1417–1427
30. Amiri, K., Gibson, G.A., Golding, R.: Highly concurrent shared storage. In: Proc. the 20th Intl. Conf. on Distributed Computing Systems, IEEE (April 2000) 298–307
31. Golding, R.: The Palladio access protocol. Technical Report HPL-SSP-99-1, Hewlett-Packard Laboratories, Storage Systems Program (November 1999)
32. Golding, R., Borowsky, E.: Fault-tolerant replication management in large-scale distributed storage systems. In: Proc. the 18th Symp. on Reliable Distributed Systems, IEEE (October 1999) 144–155
33. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) protocol version 2. IETF RFC 5246 (August 2008)
34. Atkins, D., Austein, R.: Threat analysis of the Domain Name System (DNS). IETF RFC 3833 (August 2004)
35. DARPA Tactical Technology Office: System F6. `http://www.darpa.mil/TTO/Programs/sf6.htm`, retrieved 26 March 2009