AppBackplane is an application framework that supports multiple application architectures ; provides a scheduling system that supports continuous and high loads, meets critical path timing requirements, supports fair scheduling amongst priorities; is relatively easy to program; supports higher degrees of parallelism than can be supported with a pure tasking model.

In this paper Com refers to a general messaging infrastructure that implements subscribe and publish functionality and remote method invocation. The target environment is C++ running on an embedded OS. Similar ideas may be able to be more cleanly expresses in other languages like Java.

Introduction

AppBackplane is an application framework that:

- Is relatively easy for developers to understand and implement.
- Supports static registration of names spaces. The goal is for all name spaces to be available after system modules are constructed.
- Supports multiple architectures . Applications can map to one or a pool of threads, they can be event oriented, or they can be synchronous, or they can use a SEDA pipeline type model. Multiple applications can multiplex over the same queues and threads. Applications can have multiple pieces of work outstanding at the same time.
- Supports large scale parallelism so peer-to-peer protocols can be used without radical increases in thread counts.
- Uniformly support Com, peer-to-peer, timer, and other frameworks.
- Provide a scheduling system that:
    - Supports continuous and high loads.
    - Performs work in priority order.
    - Meets critical path deadlines.
    - Does not cause watch dog timeouts.
    - Throttles by CPU usage. Once a certain amount of CPU is used the application will give up the processor.
    - Has a configurable number priority queues.
    - Has a quota of operations per queue.
- Enable request throttling total number of requests per appplication.

- Ensures operations on an object happen in order.
- Has work for an object be performed at the highest priority of outstanding requests.
- Ensures work on different objects can happen at the same time.
- Ensures one work request for an object is completed before another work request.
- Support objects modeled as state machines.
- Supports a structure that allows applications to be System Modules.
- Data structures are simple so they can be in the critical path.
- Keep various statistics on operations.

We want to make it easy for applications to select an architecture . It should be relatively simple to switch between 1-1, 1-N, M-N, event, and parallel state machine architectures. There are some additional system issues, like using static mappings that we also want to provide if we can.

Conceptual Model

In general the AppBackplane approach combines the event scheduling with parallel state machines using a thread pool (1 or more threads).

- An application derives from an AppComponent.
- AppComponents plug into and share an AppBackplane. This means one or more applications can transparently work over the same AppBackplane. This is how multiple architectures are supported.
- Work is generally asynchronous so that multiple AppComponents can multiplex over the same AppBackplane. It is not required that work be performed asynchronously, but if it is not then it is impossible to guarantee latency.
- AppBackplane has a configurable number of priority queues, called ready queues, for scheduling objects with work to do. Applications must agree on the meanings of the priorities.
- Each ready queue has a quota of the number of items that can be serviced at a time before the next priority queue must be serviced.
- AppBackplane keeps a wait queue for an object waiting for a reply. Once the reply comes in the object is moved to the ready queue.
- AppBackplane contains one or more threads blocked on a semaphore waiting for work to do. Threads can be added or deleted.
- AppBackplane is the event scheduler.
- Each object maintains a queue of work it must perform. Operations are queued to and serviced by objects. This is how we make sure operations on an object are performed in order.
- An application can work on more than one request simultaneously without creating more threads.

- All operations are in the form of Actions. This allows operations from different sources to be treated generically.
- Work is serviced in priority order. It is the object that has the priority.
- Work for an object is processed in order.
- Com and other infrastructure components are abstracted in such a way that work can be converted to Actions and that registration specifications are complete at construction.
- Com registration and deregistration will happen automatically, similarly for other services.
- AppComponent and AppBackplane are modules so they can be used as System Modules.
- AppComponent is a module so they can be in the system state machine.
- AppBackplane is a AppComponent so it can have its own operations.
- AppBackplane is a policy domain. All AppComponents sharing an AppBackplane must agree to abide by the same rules regarding latency, etc.
- AppBackplane can be a system module and move all its contained components through the system state machine.
- AppBackplane can load balance between available objects implementing the same operations.
- Each event is timed to guage how long it took. This data are used to calculate the percentage of CPU used per second which is used to limit the amount of CPU used by the backplane in a second.
- Events are mapped to objects which are queued to ready queues in the thread of the caller. If an event can not be dispatched quickly, by message type or other data in the message, then event should be queued to a **digester** object at a high priority so the event can be correctly prioritized based on more complicated calculations.
- The scheduling algorithm supports multiple priorities so high priority work gets process first, quotas at each priority for fairness, and CPU limitation.
- Com and other protocol registrations are moved into the backplane.
- Registration information for all protocols is available after backplane construction so we can support static mappings .
- A warning if issued if any event takes longer than a specified threshold to process.
- Work is scheduled by the backplane so we can keep a lot of statistics on operations.
- Many parallel activities can happen at the same time without a correspondingly large increase in thread counts.
- When the AppObject is in the Wait Queue waiting for reply, more requests can be queued to that AppObject.

The basic idea is to represent an application by a class that separates application behaviour from threading. Then one or more applications can then be mapped to a single thread or a thread pool. If you want an event architecture then all applications can share a single thread.

If you want N appplications to share one thread then you can instantiate N applications, install them into an AppBackplane, and they should all multiplex properly over the single thread.

If you want N applications to share M threads then you can create an AppBackplane for that as well.

Locking is application specific. An application has to know its architecture and lock appropriately.

Increased Parallelism Without Large Thread Count Increases

One of the motivations begind AppBackplane is to be able to support higher degrees of parallelism without radical increases in thread counts.

What this allows us to do is have more peer-to-peer protocols which is a good way to support scalability.

For example, if multiple applications on a node want to talk with 30 other cards using a peer-to-peer approach, then 30 threads per application will be used. This adds up to a lot of threads in the end.

Using the backplane all the protocols can be managed with far fewer threads which means we can expand the use of peer-to-peer protocols.

Bounding Process Times

If an application is using an event or parallel state machine model it is responsible for creating bounded processing times.

This means that whoever is doing work must give up the CPU so as to satisfy the latency requirements of the backplane. If, for example, an application expects a max sheduling latency of 5ms then you should not do a synchronous download of 1000 records from the database.

How System Modules are Supported

The backplane needs to work in the system module infrastructure. Applications need to be dependent on each other yet still work over the same backplane.

We need:

- AppComponents in the system state machine to be driven by the system state machine.
- Non system state machine components in the backplane should have their state machines driven by the backplane.
- When testing we need the backplane to drive the state machines of all components.

As appplications derive from AppComponents this means AppComponent object must be installable into dependency tables.

AppComponent derives from Module so it can be in the system module dependency table. The MoveTo? method in AppComponent calls methods that create an AppAction for each state transition. The actions implementing the state transitions are scheduled into the backplane.

For applications not in the system state machine and for testing, we need a way to have all the components in the backplane driven through their system state machine.

AppBackplane is also a module and has convenience methods like JumpToPrimary?, JumpToSecondary?, and JumpToExit? for testing. All components have their state machines driven by the backplane when these methods are used.

AppBackplane is capable of driving all its components through the system state machine when its state machine is moved.

Each component specifies if its state transitions are driven by the system state machine or the backblane.

Dispatching Work to Ready Queues

Requests like Com messages come in from a thread different than the thread in the backplane that will execute the command. A dispatch step is required to dispatch the work to an object and then to a ready queue. Dispatch happens in the thread of the caller.

Dispatching in the caller's thread saves an intermediate queue and context switch.

If the dispatch can happen quickly then all is right with the world. Dispatching can be fast if it is based on the message type or data in the request.

Sometimes dispatching is a much more complicated process which could block the caller too long.

The strategy for getting around blocking the client is to queue the operation to a high priority **digester** object.

The digester object runs in the backplane and can take its time figuring out how to dispatch the work.

Objects Considered State Machines

Objects may or may not be implement as FSMs. An object's operations may all be stateless in which case it doesn't care. Using a state machine model makes it easier to deal with asynchronous logic.

Parallelism

Parellelism is by object and is made possible by asynchronous operation. This means you can have as many parallel operations as there are objects (subject to flow control limits).

The idea is:

- Operations are queued to objects.
- Any blocking operation is implemented asynchronously allowing other operations to be performed. Latency is dependent on cooperative multitasking.
- The reply to an operation drives the state machine for the object forward.

An application can have N objects capable of performing the same operation. The AppBackplane can load balance between available objects.

Or if an operation is stateless then one object can handle requests and replies in any order so the object does not need to be multiplied.

It may still be necessary to be able to interrupt low priority work with a flag when high priority work comes in. It is not easy to decompose iterators to an async model. It may be easier to interrupt a loop in the middle.

Per Object Work Queue

The proper queue seems to be per object rather than keeping requests in a single Actor queue.

Without a per object queue we won't be able to guarantee that work performed on an object is performed in order.

The Actor queue fails because eventually you get to requests in the queue for an object that is already working on a request. Where do the requests go then?

In an async system a reply is just another message that needs to be fed to the FSM. There are also other requests for the same object in the queue. If the reply is just appended to the queue then that probably screws up the state machine because it's not expecting a new request. So the reply must be handled so that it is delivered before any new requests. This isn't that obvious how this should be done.

All Operations are Actions

The universal model for operations becomes the Action. The AppBackplane framework derives a base class called AppAction from Action for use in the framework.

All operations must be converted to AppActions by some means.

The reasoning is:

- We need a generic way to queue operations and for the work scheduler to invoke operations.
- Through the use of a base class an Action can represent any operation because all the details can be stuck in the derived class.
- Actions can be put in a memory pool so they don't have to be dynamically allocated.

Requests are converted to actions in the thread queueing the request. The actions are added to the object that is supposed to handle them. The Action/conversion approach means that the type of a message only has to be recovered once. From then on out the correct type is being used. Parallelism can be implemented by the application having more than one object for a particular service.

AppEndpoints Create AppActions

There has to be a way to convert operations from different protocols (Com, timer, DataGrid, etc) to actions so they can be scheduled and executed over the backplane. AppEndpoints are this way.

AppEndpoint objects are used as to convert requests and replies to AppAction objects. AppEndpoint derives from MsgHandler so it can be the endpoint in Com and other protocols.

ComImplement?, for example, is an AppEndpoint for handling incomming Com requests. When a request comes on a namespace Com calls ComImplement's HandleMsg?. HandleMsg? then calls CreateImplementationAction? which is overridden by a derived class that knows how to convert the request to an AppAction. The class derived from AppAction knows how to implement the operation.

AppAction specifies which AppObject is should be schedule to. Then AppObject is scheduled in the backplane.

Ready and Wait Queues

The ready and wait queues take the place of the single Actor queue. AppBackplane is the event scheduler in that it decides what work to next. AppComponents are multiplexed over the AppBackplane.

- An object cannot be deleted if it is the wait queue.
- Aggregation is not performed in the queues. The correct place for aggregation is in the senders. Most every request expects a reply, so aggregation usually won't work.
- The total number of requests is counted so that we can maintain a limit.
- The number of conccurent requests is limitted.

Handling Replies

An object can have only one outstanding reply so we don't need dynamic memory to keep track of an arbitrary amount of outstanding requests.

If an object is expecting reply the object is put on the wait queue until the reply comes in. The reply is queued first in the object's work queue. The object is then scheduled like any other object with work to do.

Client Constraints

Clients can not send multiple simultaneous requests and expect any order. Order is maintained by a client sending one request to a service at a time and not duplicating requests.

Steps to Using AppBackplane

- Create a backplane to host components that agree to share the same policies. Policies include:
  - Number of priorities.
  - Quotas per priority.
  - CPU limit.
  - Latency policies. What is the longest an application should process work?
  - Lock policies between components.
  - Number of threads in the thread pool. This addresses lock policies.
- For each application create a class derived from AppComponent and install it into the backplane.
- Determine how your application can decomposed asynchronously.
- Install the backplane and/or components in the system modules if necessary.
- In each component:
  - Implement virtual methods.
  - Create AppEndpoints for each operation in each protocol.
  - Create AppActions for implementing all operations.
  - Decide which objects implement which operations.
  - Decide which operations are implmented in the AppAction or in the object.
  - Decide if objects represent state machines.

Class Structure

```
          +-- ISA ----[Module]
          |
[AppBackplane]--HAS N ----[AppComponent]
```

```
      |
      +-- HAS 1 ----[AppScheduler]


[AppScheduler]-- HAS N ----[AppWorkerThread]
     | |
     | +-- HAS N ----[WorkQueue<Ready>]
     |
     +-- HAS 1 ----[WorkQueue<Wait>]


      +-- ISA ----[Module]
      |
[AppComponent]-- HAS N ----[AppObject]
      |
      +-- HAS N ----[AppEndpoint]


[AppWorkerThread]-- ISA ----[LnTask]

[AppObject]-- HAS N ----[AppAction]

[AppAction]-- ISA ----[Action]

[AppEndpoint]

[MsgHandlerEndpoint]-- ISA ----[AppEndpoint,MsgHandler]

[ComImplement]-- ISA ----[MsgHandlerEndpoint]

[PsAdd]-- ISA ----[ComImplement]
```

Com Processing Example

This section covers details about how Com operations are handled over the backplane.

Com Implement Processing

This is an example how Com requests are handled using AppBackplane.

- An application has created its AppComponent and created ComImplements? etc for each operation. These operations were registered during construction using AppComponent::AddOperation.
- Operations are derived from MsgHandlerEndpoint? which is a MsgHandler. It supplies the namespace and in what system states the operations should be registered.
- To implement the Ps add operation, Ps would create PsAddOp? that derives from ComImplement?.
- At the appropriate system state the backplane will register with Com.
- Com requests for the operation are routed by Com to the MsgHandlerEndpoint?.
- MsgHandlerEndpoint? calls CreateImplementationAction? so the derived MsgHandlerEndpoint? class can create an action that knows how to implement the operation.
- During the CreateImplementationAction? process the MsgHandlerEndpoint? must have figured out which object the action should be queue to. It will do this using information from the request or it will have predetermined objects already in mind. The AppComponent and AppBackplane are available.
- Objects capable of processing AppActions must derive from AppObject.
- The backplane is asked to schedule the AppAction.
- The AppAction is queued to the AppObject specified in the AppAction.
- The AppObject is put on the priority queue specified by the AppObject.
- An AppWorkerThread? is allocated from the idle pool.
- The AppWorkerThread? thread priority is set to that indicated by the AppObject.
- The AppObject is given to the AppWorkerThread? when it gets a chance to run.
- If a worker is not available the work will be scheduled later.
- The AppWorkerThread? is put on the active pool.
- The first AppAction in the queue is dequeued and the action's Doit method is invoked.
- How the action accomplishes its task is up to the action. All the logic for carrying out the operation may be in the action. The action may invoke an event on a state machine and let the object take over from there. The operation may:
    - do its work and reply immediately
    - do some work and block waiting on an asynchronous reply
    - perform a synchronous call
- If the object needs to send out more requests it will send the request and then call DoneWaitForReply? so it will be put in the wait queue waiting for a reply. See Com Invoke Processing.
- If the object is done it calls Com's Reply to send the Reply back to the sender. Then it calls DoneWaitForMoreWork? to schedule it for more work.

Com Invoke Processing

- A class is derived from ComInvoke? to represent the invovation. It is capable of transforming the reply into an action. The ComInvoke? is given to Com invoke as the MsgHandler. The HandleMsg? calls CreateImplementationAction? to create the

action to handle the reply.

- The reply uses the context ID to contain the pointer to the object that should handle the reply.
- Replies are inserted before requests so that the reply can be processed before requests. An object can have only one outstanding reply at a time.
- An object with a reply is put in the ready queue so the reply can be processed by the object.
- The object is then scheduled.

Com Publish and Subscribe Handling

Handle similarily to implement and invoke.

Scheduling Algorithm

AppBackplane implements a scheduling layer on top of OS thread scheduling. An extra scheduling layer is necessary to handle fairness, low latency for high priority work, and throttling in a high continuous load environment.

The scheduling algorithm is based on:

0. integration period - a period, 1 second by default, over which CPU
   utilization is calculated. Quotas are replenshed. When this timer
   fires we call it a "tick."
1. priority queues - work is performed in priority order. Priorities are
   0..N where 0 is the highest priority. The number of queues is configurable.
2. thread pool - more than one thread can be active at a time. The thread
   is scheduled and is run at the priority set by the work being done.
   The number of threads in the pool is configurable. Thread can be
   dynamically added and deleted.
3. CPU limitation - each backplane can use only so much of the CPU
   in an integration period. If the period is execeeded work is stopped.
   Work is started again on the next tick.
4. quotas - only so much work at a priority is processed at a time. Quotas
   are settable by the application. By default quotas are assigned using
   the following rules:
   1. The highest priority has an unlimitted quota.
   2. The next highest priority has a quota of 100.
   3. Every priority then has 1/2 have the previous higher priority until
      a limit of 12 is reached.
5. priority check - after each event is processed the scheduler starts
   again at the highest priority that has not used its entire quota.
   The idea is work may have come in for higher priority work while

lower priority work was being processed. A priority may run out of work before its quota has been used so there may be more work that can be done at a priority.

6. virtual tick - If all the CPU hasn't been used and all quotas at all priorities have been filled, then a virtual integration period is triggered that resets all quotas. This allows work to be done until the CPU usage limit is reached. Otherwise the backplane could stop processing work when it had most of its CPU budget remaining. The CPU usage is not reset on a virtual tick, it is only reset on the real tick.

Backplanes are Not Organized by Priority

It is tempting to think of backplanes as being organized by thread priority. This is not really the case. Any application has a mix of work that can all run at different thread priorities.

You probably don't wan't to splat an application across different backplanes, though technically it could be done.

The reason is backplanes are more of a shared data domain where latency contracts are expected to be followed. Different backplanes won't share the same latency policies, lock polocies, or ready queue priorities and quotas.

Thus backplanes can't be thought of as able to cooperated with each other.

CPU Throttling

CPU throttling is handled by the scheduler.

Request Throttling

Requests are throttled by having a per component operation limit. When a component reaches its limit and a new operation comes in then the component is asked to make room for new work, if the component can't make room then the request is failed with an exception.

The client can use the exception as a form of backpressure so that it knows to wait a while before trying the request again.

Priority Inheritence for Low Priority Work

Low priorty work should have its priority raised to that of the highest work outstanding when the low priority work is blocking higher priority work that could execute.

The machism needs to be determined. Thread local? A global static? Clearly each backplane can knows its highest priority work that can run.

The reasoning is...

Work is assigned to threads execute a particular priority.

Thread priorities are global. We must assign thread priorities on a system wide basis. There doesn't seem to be a way around this.

Each backplane has priority based ready queues that are quota controlled. High priority work that hasn't met its quota is executed. By priority we are talking ready queue priority, not thread priority. Thread priority is determined by the work.

Ready queue priorities and quotas are primarily determined by the backplane. They are ultimately constrained by a CPU limit on the backplane.

Lower priority work is scheduled when there is no higher priority work or higher priority work has met its quota.

When lower priority work has been assigned to a thread higher priority work can come in.

The lower priority work's task may not be scheduled because a higher priority task is executing elsewhere in the system. This causes the lower priority work not the be scheduled and to not get a chance to run. In fact, it can be starved.

The lower priority work is causing the higher priority work not to execute, even if the higher priority work would run at a task priority higher than the currently running task that is blocking the lower priority work from running.

By upping the task priority of the lower priority work's task to that of the higher priority work's task priority we give the lower priority work a chance to run so it can complete and let the higher priority work run.

The lower priority work can run to completion or it can use global flags to know if it should exit its function as soon as possible so the higher priority work can run.

The priority inheritance works across backplanes so high priority work in any backplane will not block on lower priority work in any other backplane. The higher priority work is present flag is global.

Even if the lower priority work doesn't not give up the critical section immediately, just giving it a chance to run will make it so the higher priority work can get scheduled sooner.

Fair sharing is implemented by a max CPU usage being assigned to each backplane.

Backplane Doesn't Mean No Mutexes

Mutexes can still be used within a backplane, but they should not be shared with threads outside the backplane because there can be no guarantees about how they are used.

Applications inside a backplane should be able to share mutexes and behave according to a shared expectation of how long they will be taken and when they will be taken.