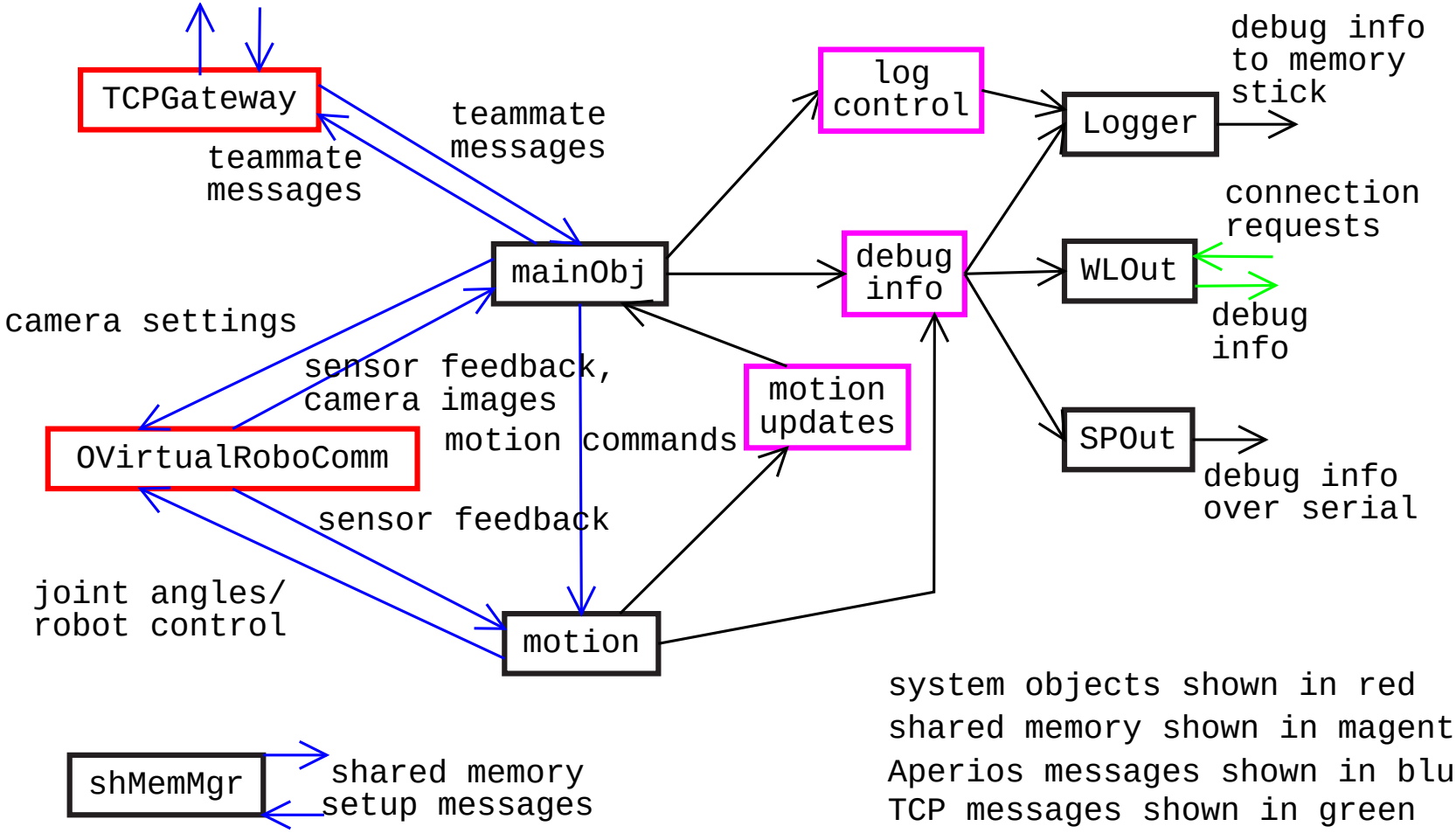
System Overview

Carnegie Mellon University

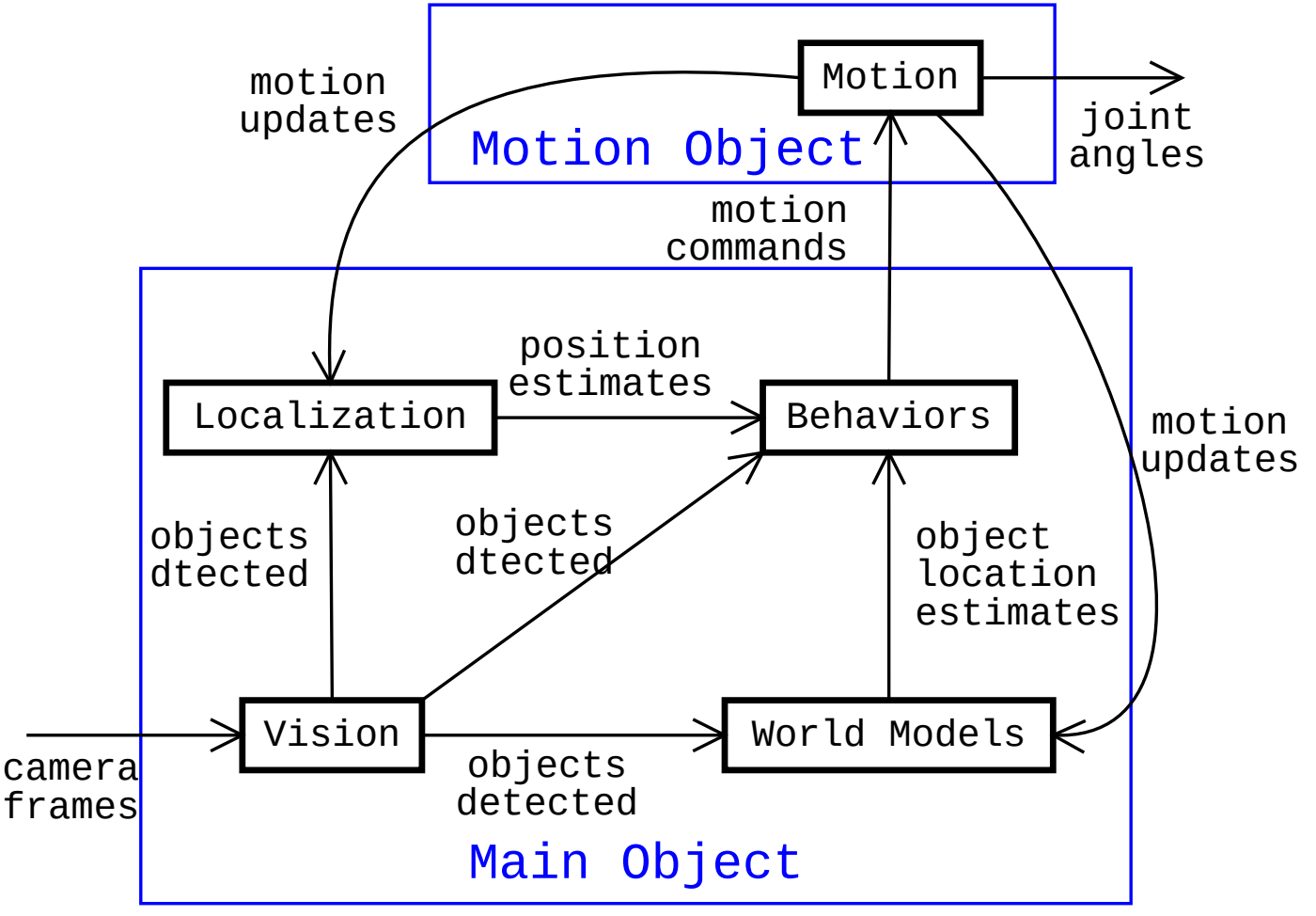
October 13, 2003

Scott Lenser
Manuela Veloso

The Big Picture



The Big Picture - Major Pieces



Aperios Objects - Object Definition

Aperios objects are made using the mkbin program on a bunch of C++ object files.

Parameters of the object are set using a .ocf file. Here is an example file used for mainObj:

```
object mainObj 16k 512k 128 cache tlb user
```

In this file:

- mainObj is the name of the object.
- 16k is the size of the stack.
- 512k is the maximum size that can be allocated in the object heap at one time.
- 128 is the object priority. Aperios uses a strict priority scheme where higher priority objects get first chance at CPU time. Lower priority objects get whatever CPU time is left over. If the CPU time is overcommitted, lower priority objects will never run.

Aperios Objects - Object Initialization

Steps in Aperios object initialization:

- The constructor for the object is called. This step happens sequentially for each object.
- The DoInit system message is sent. This step should perform the bulk of the initialization needed by the object. This step happens in an undefined order between different objects.
- The DoStart system message is sent. This step should activate the connections to other Aperios objects. This step happens in an undefined order between different objects.

Aperios Interobject Communication

- Aperios uses a publish/subscribe interobject communication mechanism.
- Publishers (called "subjects" in Aperios, think sender) are declared statically.
- Subscribers (called "observers" in Aperios) are also declared statically.

Aperios Interobject Communication

- Subjects and observers are declared in a stub.cfg file.
- The connections to make are specified in the connect.cfg file.

Show stub.cfg for MotionObject.

Show callbacks in MotionObject.

Show connect.cfg.

Interobject Communication - Messages

- AssertReady() - this call sends a message to tell the subject(s) attached to this observer that the observer is ready for more information.
- OReadyEvent - a message containing this event is sent to the subject(s) in response to an AssertReady call
- SetData(), NotifyObservers() - these two calls are used to send data from a subject to one or more observers
- ONotifyEvent - a message containing this event is sent to the observer.
- AddReference() - can be used to lock the contents of an ONotifyEvent. This is the way shared memory regions are implemented.

Debug Support Infrastructure

- There are 3 major debugging modules: SPOut, WLOut, and Logger
- SPOut provides output facilities over serial link.
- WLOut provides output facilities over waireless LAN.
- Logger provides logging facilities to the memory stick.
- All 3 modules read data from the same shared memory regions.
- There is one shared memory region for each Aperios object.

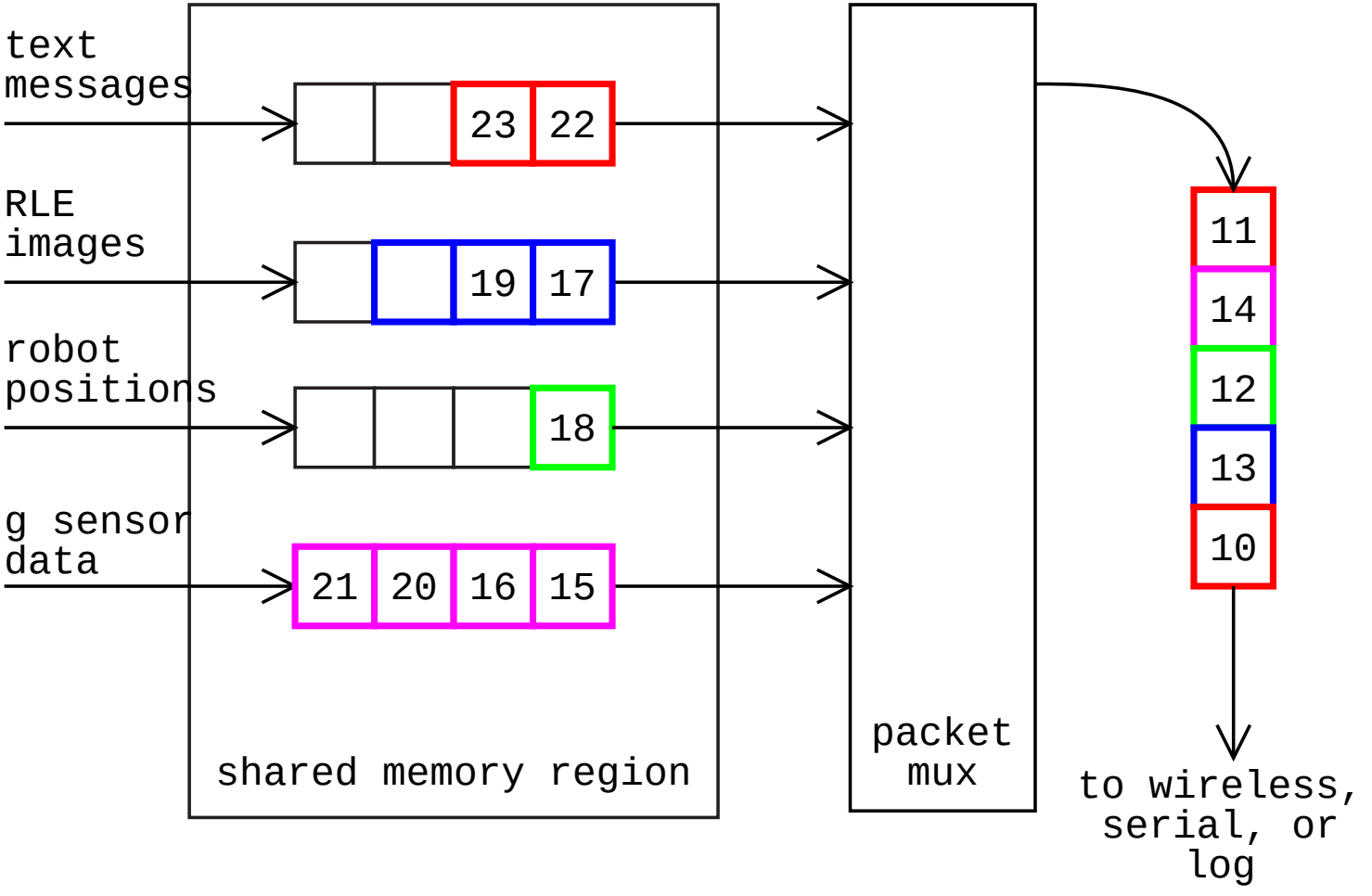
Debug Support Infrastructure

- Most of the code for interfacing with the shared memory regions is contained in agent/headers/CircBufPacket.h and agent/shared_code/CircBufPacket.cc.
- Each shared memory region contains a PacketStreamCollection.
- Each PacketStreamCollection contains a set of PacketStreams.
- Each stream stores a sequence of RobotDataPackets. Each packet contains a piece of information about the robot, usually encoded in binary. The data in packets is a compact representation of information on the robot.
- Each stream has a unique id defined in agent/headers/SPOutEncoder.h. Ids are in the range 0x82 to 0xBF.

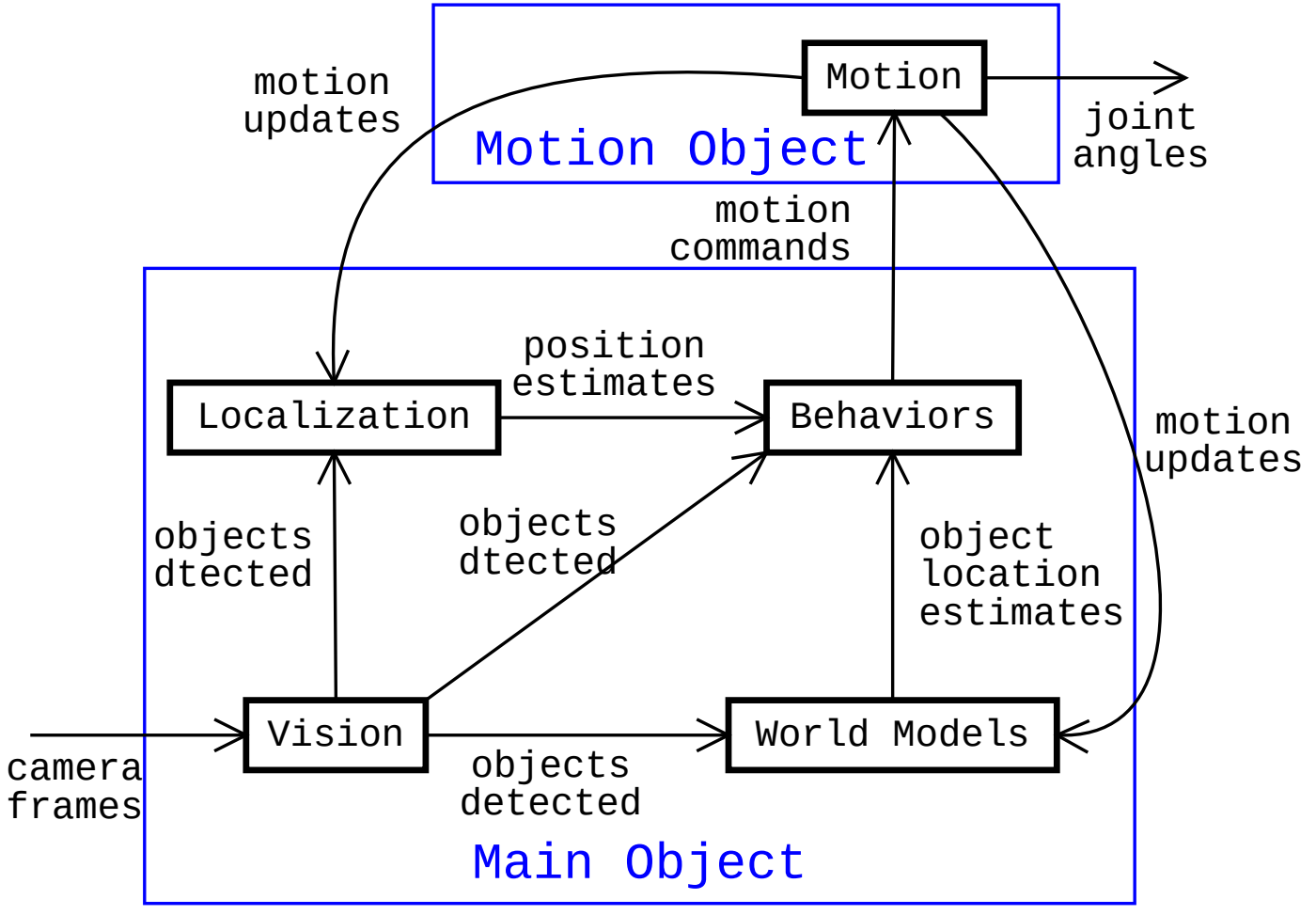
Debug Support Infrastructure

- On the receiving side of the packets, the packets from the different streams are serialized by a PacketMux (see agent/headers/PacketMux.h and agent/shared_code/PacketMux.cc).
- The PacketMux for SPOut and WLOut uses a round robin strategy. This may result in out of order packets across streams.
- The PacketMux for Logger uses a FIFO strategy.
- Packet multiplexers for SPOut and WLOut may drop packets if the bandwidth available is exceeded.

Debug Support Infrastructure



The Big Picture - Major Pieces



Motion

- You have already worked with this module.
- Responsible for generating motions requested by behaviors including walks, kicks, getups, etc. Motion commands sent via an Aperios message.
- Reports motion resulting from motions to rest of system via a shared memory region.
- Module is located in agent/Motion.
- Interface is defined in agent/Motion/MotionInterface.h.

Motion - Major Parts

All major parts located in agent/Motion/Motion.cc.

- Motion control state machine, implemented in Motion class.
- Walk engine, implemented in Trot class.
- Motion replay, implemented in MotPlayer class, driven by Getup and Kick classes.

Vision

- You have already worked with this module.
- Responsible for extracting information from camera images.
- Primary output is location of objects and radial vision output.
- Module is located in agent/Vision.
- Interface is defined in agent/Vision/VisionInterface.h.
- Object detection is done in agent/Vision/Vision.cc.
- Radial vision is done in agent/Vision/Radial.cc.

Localization

- Responsible for reporting the location of the robot in world coordinates.
- Gets input from vision about landmarks detected.
- Gets input from motion about movements of the robot.
- Module is located in agent/Localization.
- Interface is defined in agent/Localization/LocalizationInterface.h.

Localization - Code Pieces

- There are two implementations of the localization module.
- A Gaussian based implementation is in agent/Localization/GLocalization.cc.
- A newer particle filter implementation is in agent/Localization/SRL.

World Models

- A set of modules responsible for integrating information from sensors over time.
- Receive input from vision about images and from motion about movement of the robot amongst other sources.
- Most modules located in agent/WorldModel.

World Models

- The world model (defined in agent/WorldModel/WorldModel.cc) tracks the position of the ball and goals relative to the robot.
- The bot model (defined in agent/Behaviors/BotModel.cc) tracks the position of robots relative to the robot.
- The local model (defined in agent/WorldModel/LocalModel.cc) tracks the position of objects relative to the robot using input from the radial vision.
- The posture detector (defined in agent/WorldModel/PostureDectector.cc) detects when the robot has fallen over.
- The stuck detector (defined in agent/WorldModel/StuckDetector.cc) detects when the robot is hung up on a wall or another robot.

Behaviors

- Responsible for controlling what the robot does.
- Primary output is commands to the motion module.
- Takes input from every other module (Vision, Localization, World Models, Motion).
- Module is located in agent/Behaviors.
- Interface for is located in agent/Behaviors/Behavior.h.

Behaviors - Major Parts

All major parts are located in agent/Behaviors.

- FeatureSet.cc provides a summary of all inputs into behavior system.
- state_machine.h defines a utility class for state machines. Most behaviors are built upon this class.
- BeLowLevel.cc defines a bunch of utility behavior routines.

The Event System

- Based upon EventProcessor abstraction defined in agent/Main/Events.h.
- EventProcessors are organized into a directed acyclic graph.
- Events are inserted at the left(start) edge of the graph in response to system messages.
- Motion commands are extracted from the right(end) edge of the graph.
- EventProcessors are connected using a publish/subscribe model.

Event Processors

- Each EventProcessor defines two main functions.
- update() is called on this EventProcessor whenever the processors it has subscribed to have new data available.
- The update() function is passed a list of events that caused this update. It returns true if this processor has new data available. This function is guaranteed to be called when the subscribed-to processors have new data available.
- The get() function is used to get the output of this EventProcessor. This function is only called if the output of this processor is needed. This function often calculates the output only on request. The function should cache the value to avoid excessive computation.

Event Processors

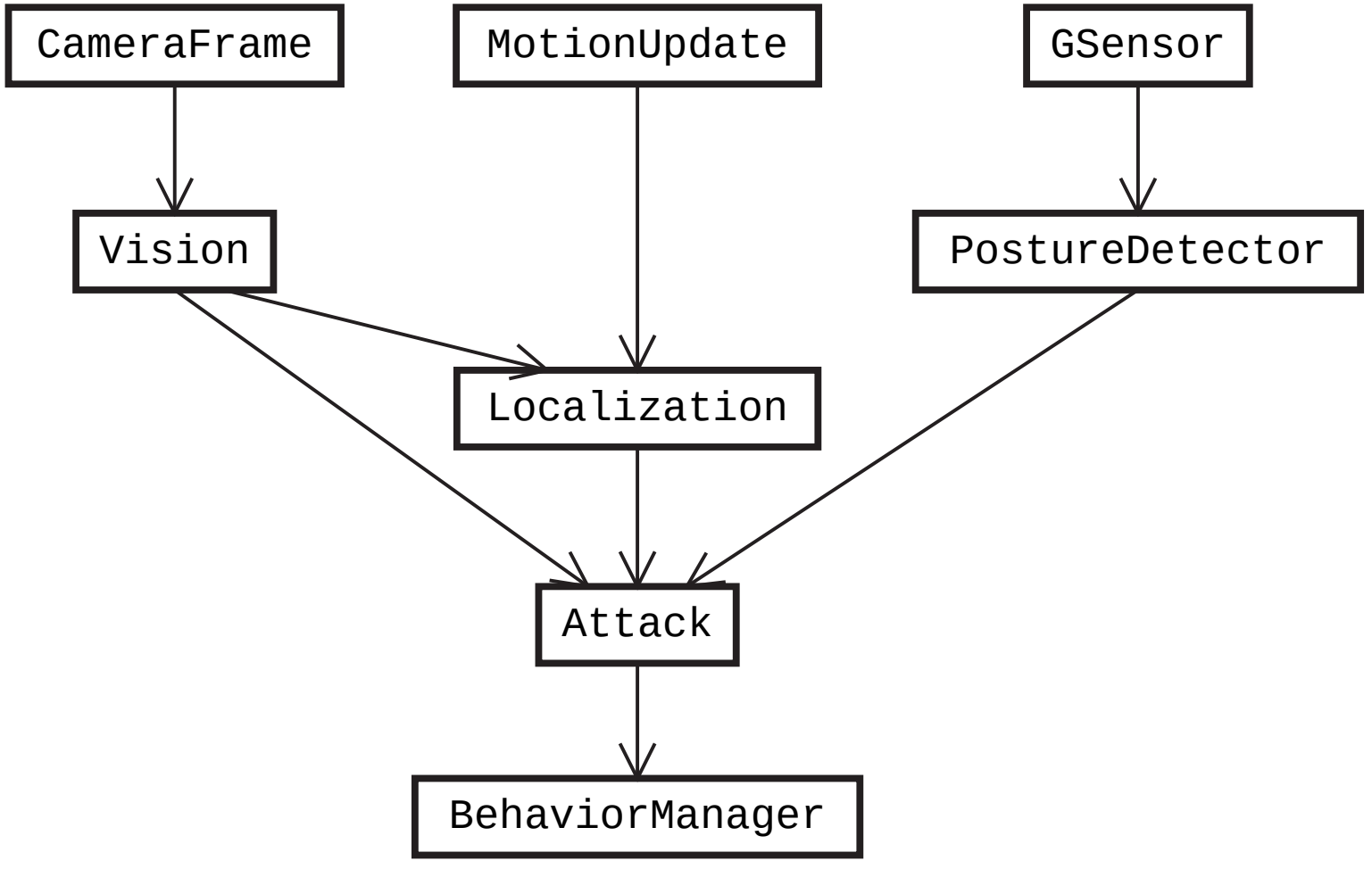
- The EventCacheTracker class can help you keep track of a cache for an EventProcessor. It is defined in agent/Main/Events.cc.

- Behaviors that can be run from run.cfg are called IndependentBehaviors because they require no inputs other than sensors. They must satisfy the interface in agent/Behaviors/Behavior.h.

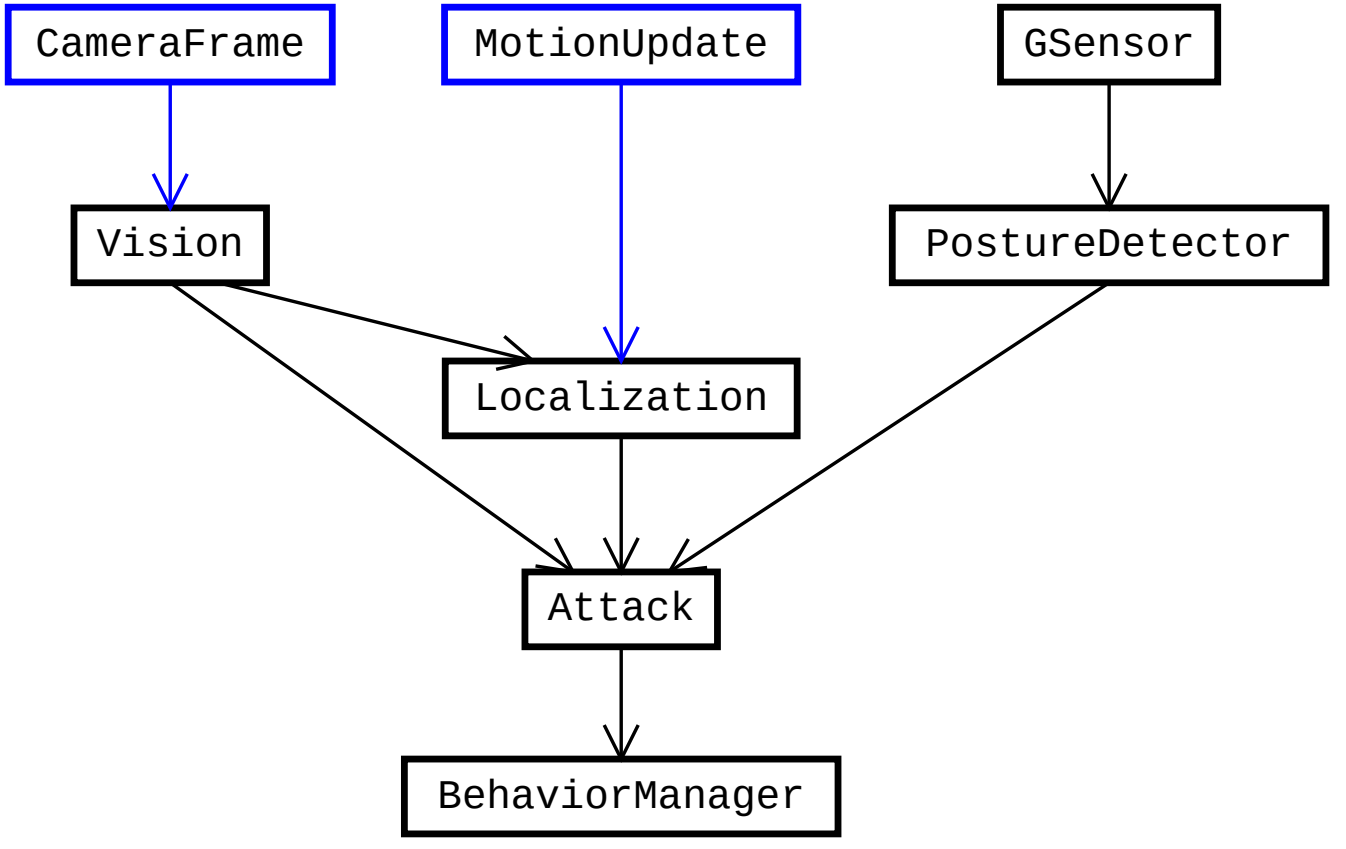
The Event System

- The event system is activated by RobotMain in agent/Main/RobotMain.cc.
- Events are injected into the system through system EventProcessors defined in agent/Main/SystemEvent.cc.
- The propagation of events is handled by the EventManager class in agent/Main/Events.cc.
- The BehaviorManager class in agent/Main/SystemEvent.cc collects motion commands from behaviors and integrates them into one motion command. This EventProcessor ensures that the behaviors listed in run.cfg get instantiated and collects input from them as it becomes available.

The Event System

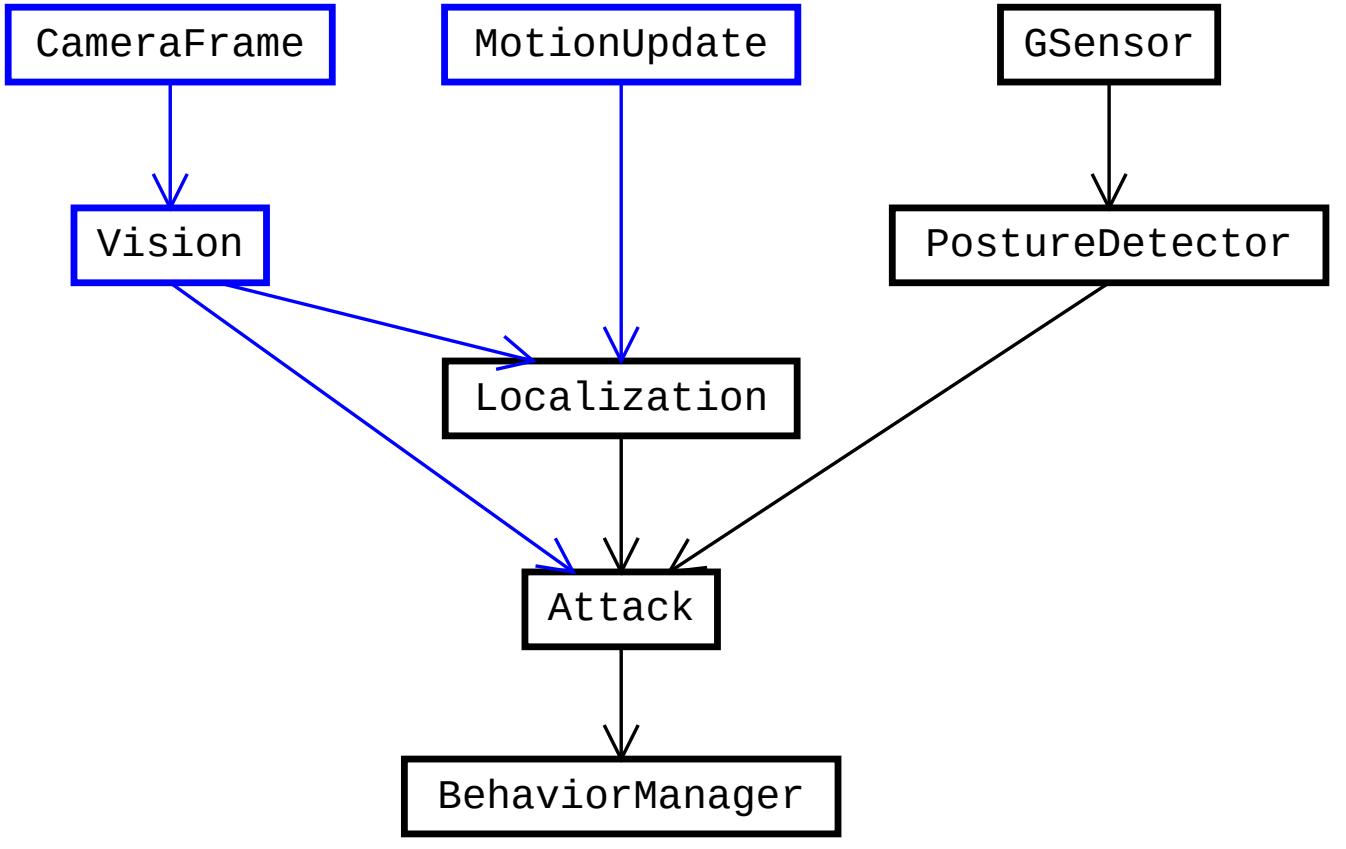


Event Propagation



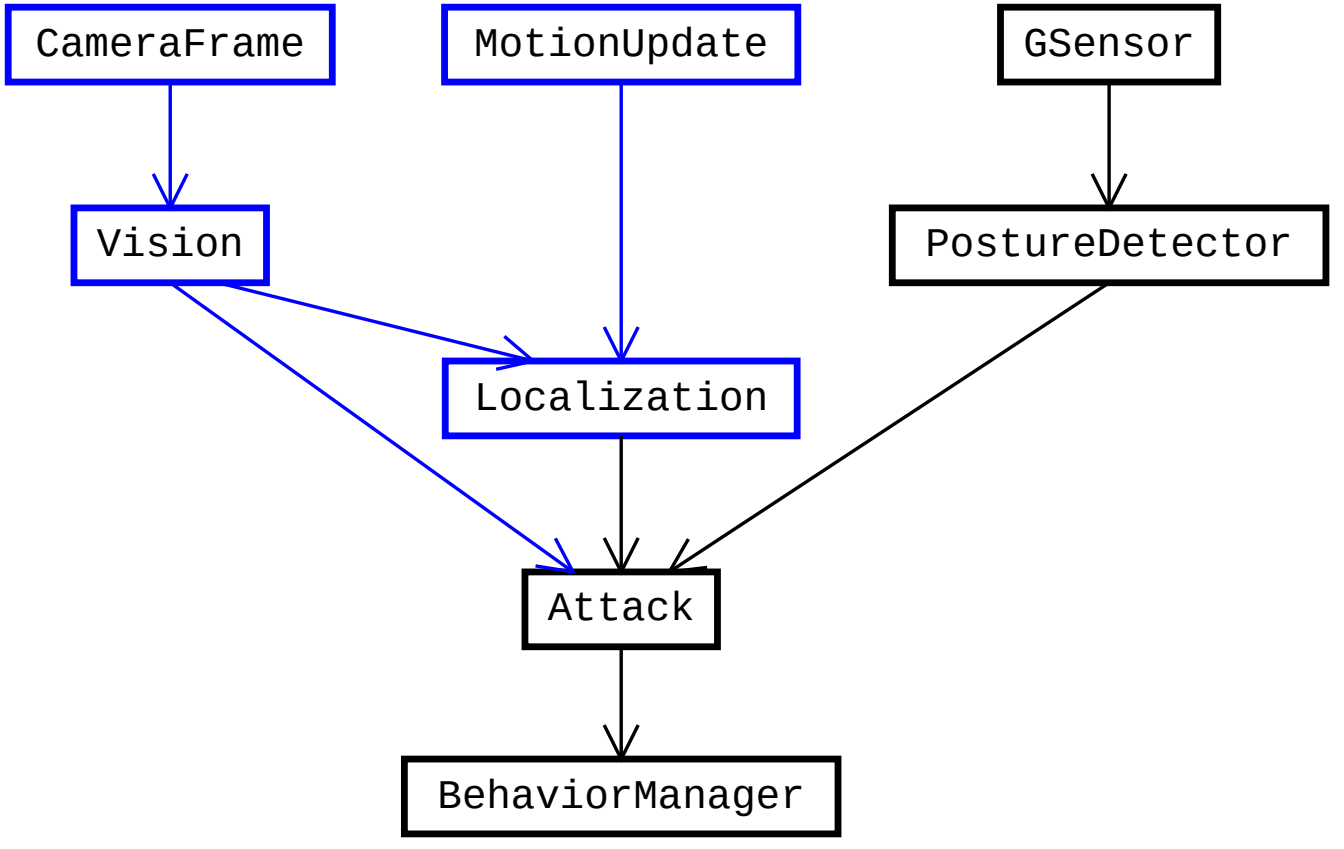
update() called or propagating
get() called or propagating

Event Propagation



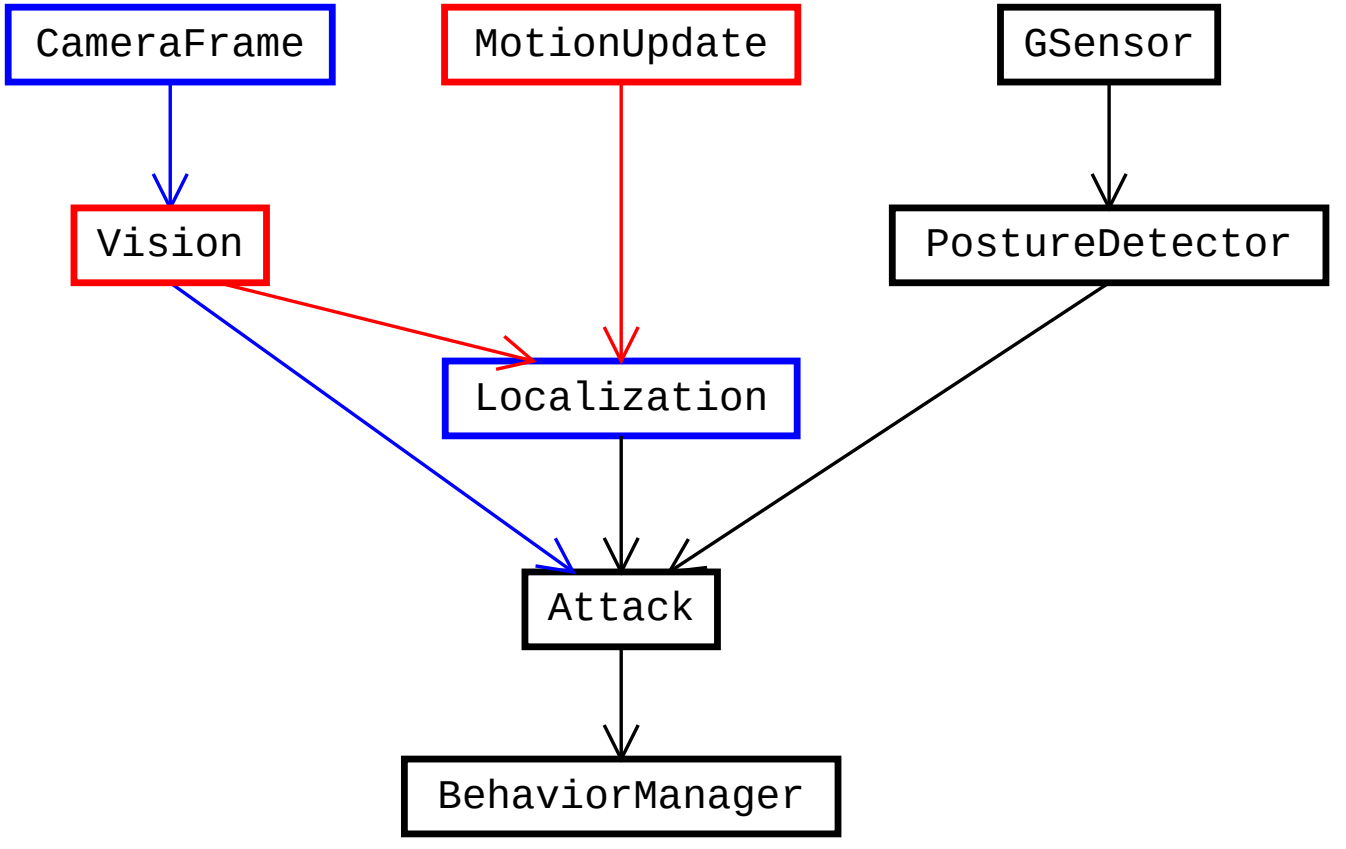
update() called or propagating
get() called or propagating

Event Propagation



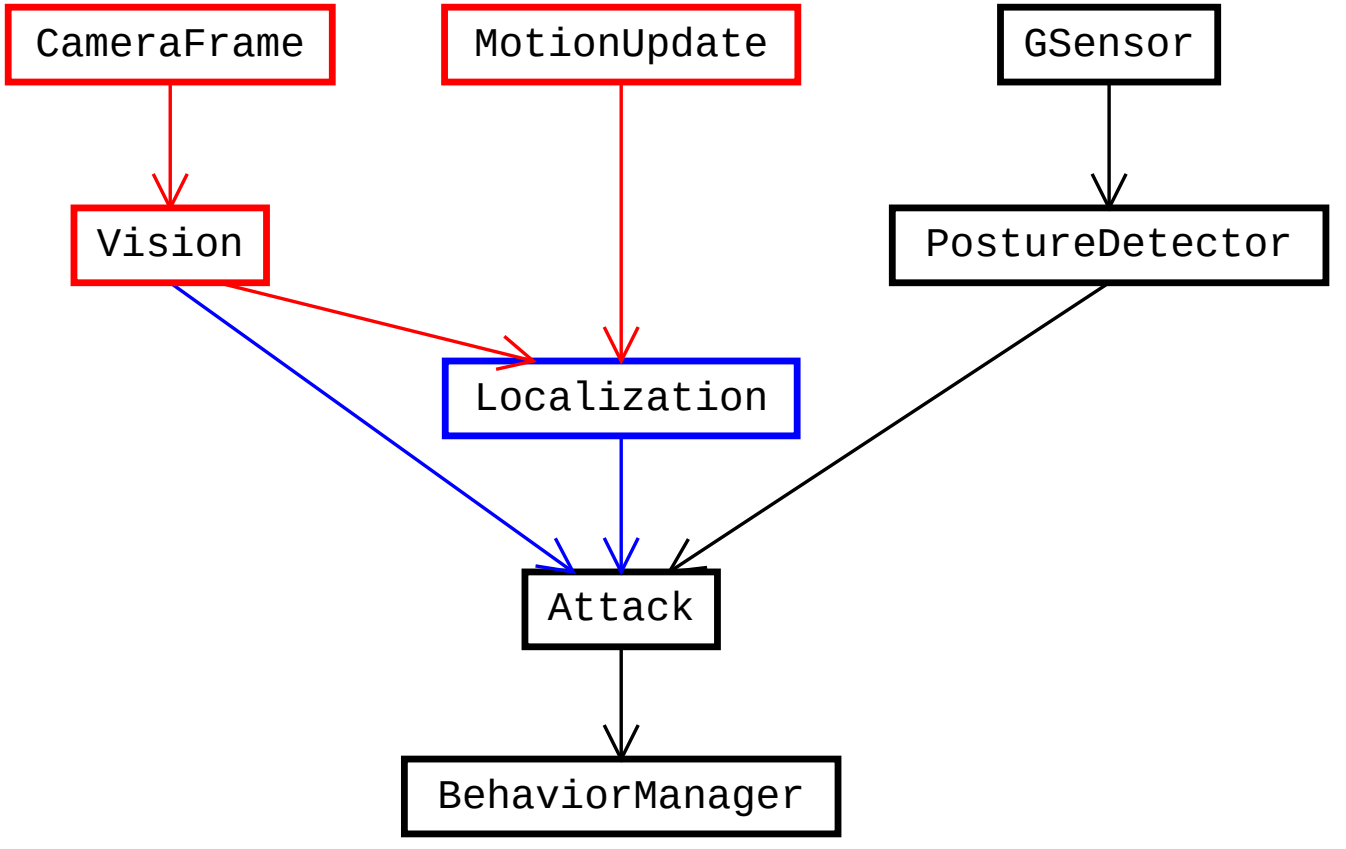
update() called or propagating
get() called or propagating

Event Propagation



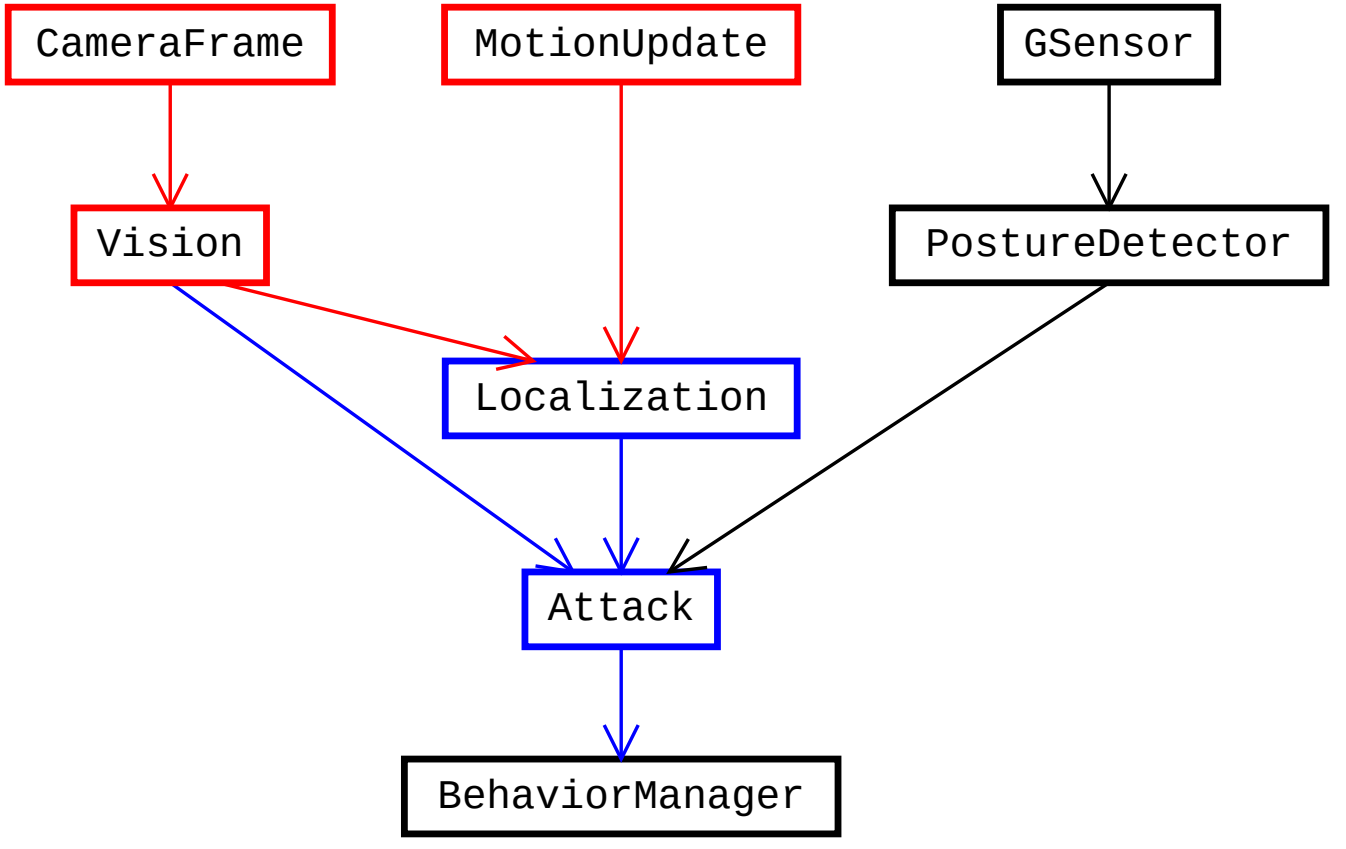
update() called or propagating
get() called or propagating

Event Propagation



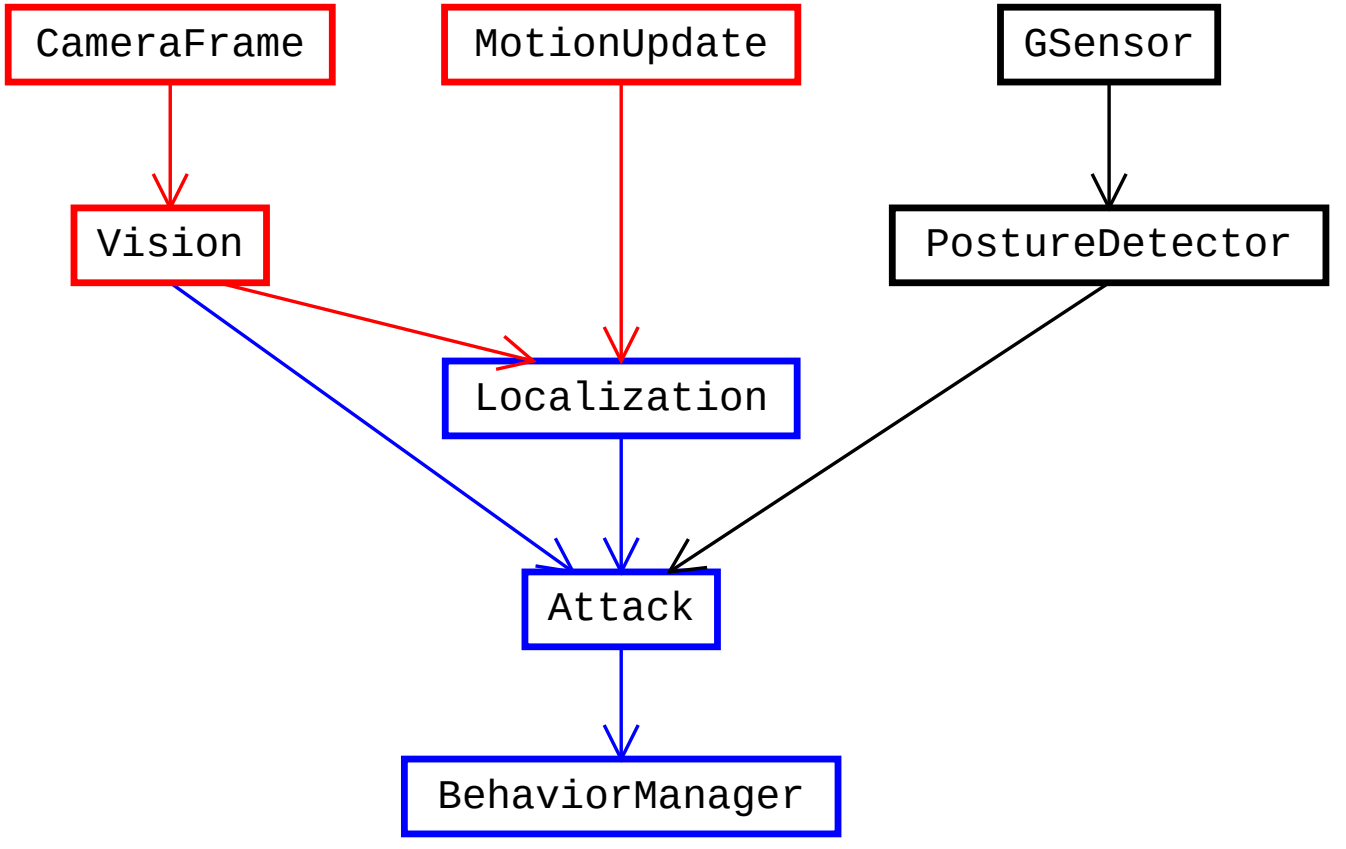
update() called or propagating
get() called or propagating

Event Propagation



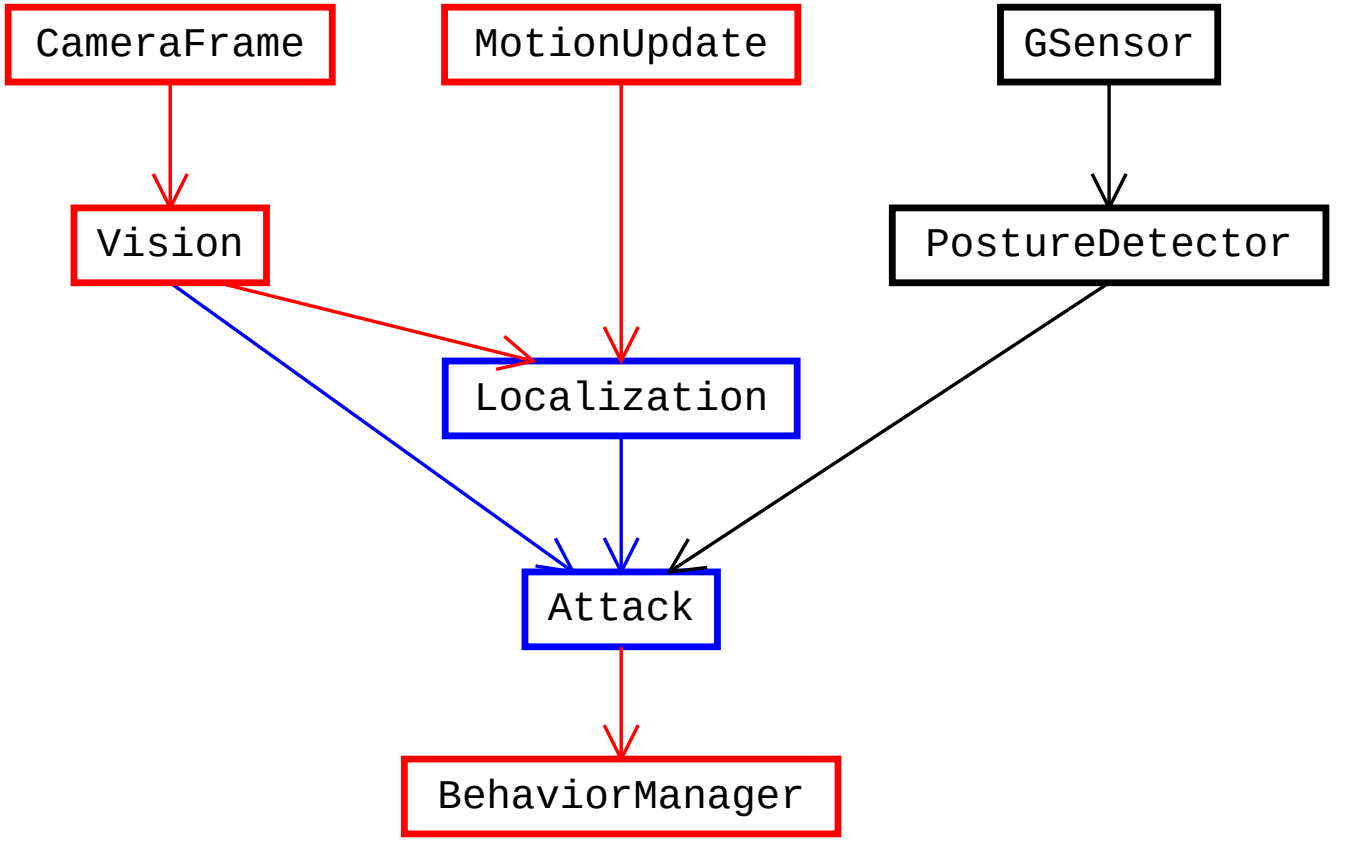
update() called or propagating
get() called or propagating

Event Propagation



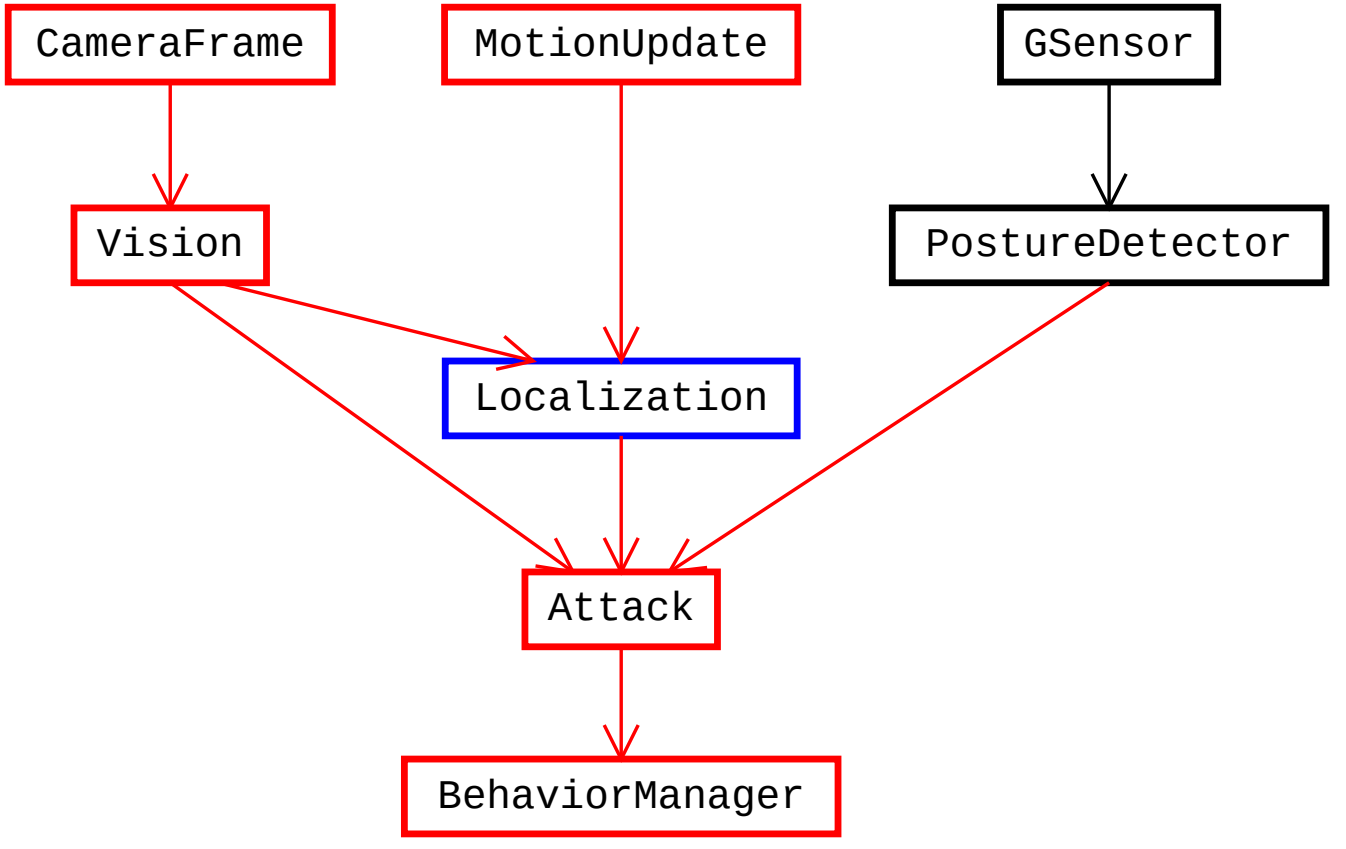
update() called or propagating
get() called or propagating

Event Propagation



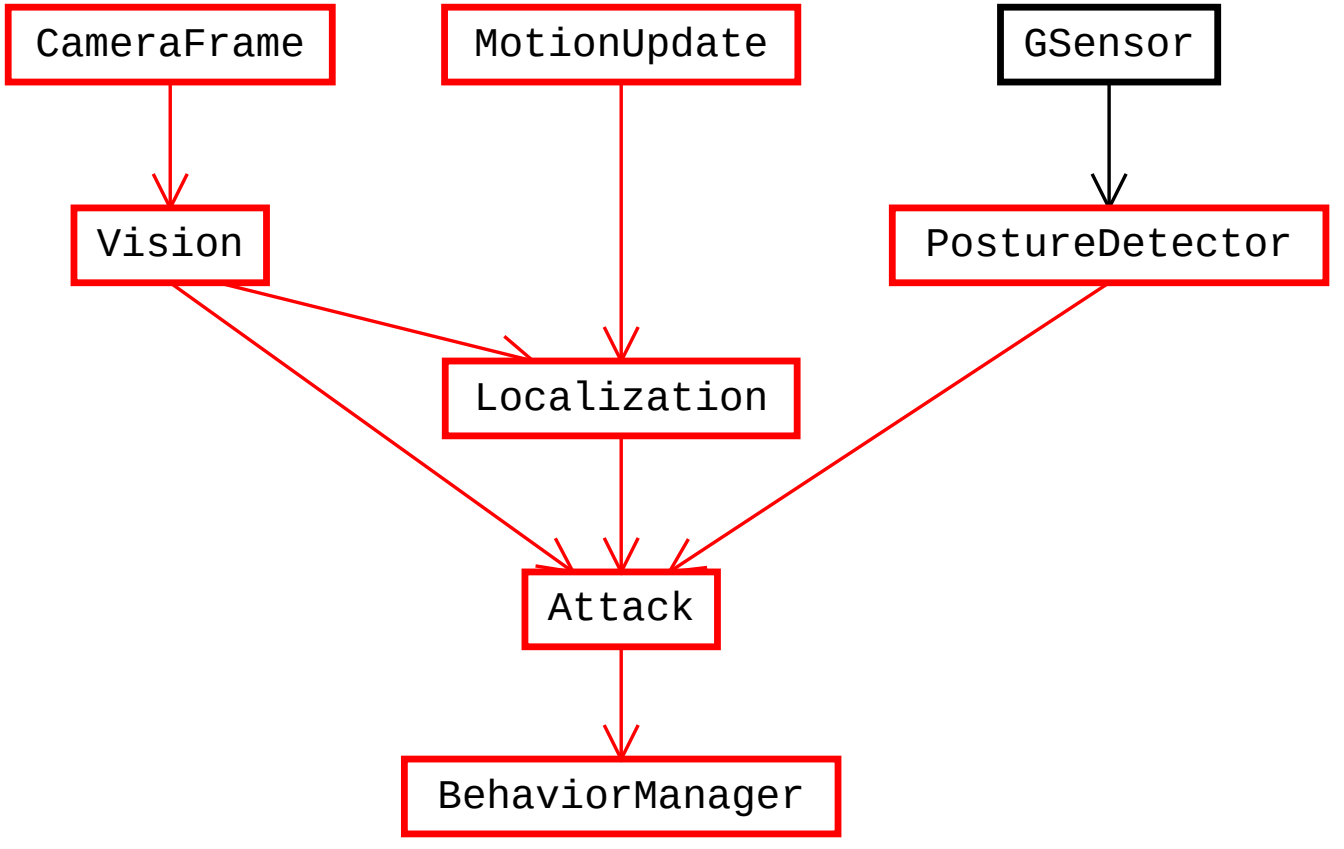
update() called or propagating
get() called or propagating

Event Propagation



update() called or propagating
get() called or propagating

Event Propagation



update() called or propagating
get() called or propagating

The Event System - Publish

- EventProcessors must be registered with the system via the REGISTER_EVENT_PROCESSOR macro. This publishes there availability. This macro can be found in agent/Main/Events.h.

- EventProcessors are created when someone asks for them by name. To enable this, all EventProcessors must define a create function that is passed to the REGISTER_EVENT_PROCESSOR macro.

The Event System - Subscribe

- Every event processor is assigned an id by the system that can be compared with those provided during an update() call to see what processors have new information available.
- An event processor's id can be queried through the getEventProcessorId() method of the EventManager class.
- The event processor itself can be accessed through the getEventProcessor() method.
- When an event processor is created, its initConnection() method is called. This should setup all the connections needed by this event processor and return true on success.
- To setup a connection, the listenEventProcessor() method of EventManager should be called.