	Aibo1		Aibo2
Boot			
	create `erouter` instance		create `erouter` instance
Setup			
	new EventRouter instance created (’`remoteRtr`’), connected to Aibo2	$\longrightarrow$	create new EventListener (’`listen1`’), receives network data from Aibo1, but events from Aibo2:`erouter`
	$\vdots$		
	Behavior (’`beh`’) registers for `(buttonEGID, HeadButOffset, activateETID)` on `remoteRtr`		
	$\downarrow$		
	`remoteRtr` sends `(erouterEGID, buttonEGID, activateETID)` (signals first listener for `buttonEGID`	$\xrightarrow{(1)}$	`listen1` subscribes to `buttonEGID`[2] from `erouter`
Usage			
			button gets pressed
			$\downarrow$
	`remoteRtr` gets button event, sets host field to Aibo2	$\longleftarrow$	`listen1` serializes[3] event, sends over network
	$\downarrow$		
	`beh` receives[3] Aibo2’s button event		
Teardown			
	`beh` is destructed, unsubscribes from `remoteRtr`		
	$\downarrow$		
	`remoteRtr` generates `(erouterEGID, buttonEGID, deactivateETID)`	$\longrightarrow$	`listen1` unsubscribes for button events from `erouter`
	$\vdots$		$\vdots$
	`remoteRtr` is destructed, closes connection to Aibo2	$\longleftrightarrow$	`listen1` destructed, unsubscribes any remainder from `erouter`

[1]Could be done either by making `remoteRtr` a subclass of EventRouter which is network-aware, or by instantiating an EventListener which subscribes to `remoteRtr`’s `erouterEGID`, so each time listeners are added or removed this separate class would handle the appropriate network communications. The former implementation has simpler information flow, but will require you to learn some EventRouter internals. The latter might be a little easier to implement, (especially given footnote 3), but is perhaps a little less elegant. (debatable)

[2]Note that with a basic implementation, although `beh` only wants head button events, `listen1` is subscribing to *all* button events. `remoteRtr` would filter the extra, so `beh` will only get what it expects, but the dropped events are wasting bandwidth and possibly CPU on Aibo2. A more advanced implementation of this process would not just send the `erouterEGID` event directly, but also send the level of specificity that is being subscribed so that `listen1`’s subscriptions directly match `beh` (plus any other behaviors accessing `remoteRtr`). This could be done by either using a new EventBase subclass for erouterEGID events to hold the addition information, (and then using the basic implementation for serialization), or by using a custom protocol for communication between `remoteRtr` and `listen1`.

[3]To handle events serialization/deserialization polymorphically, the EventTranslator (`Events/EventTranslator.{h,cc}`) class has been provided, which implements encodeEvent() and decodeEvent(). EventTranslator is itself an EventListener, automatically calling encodeEvent for any events it receives. Thus, a subclass of EventTranslator could be created to encode and send over the network with relative ease. Alternatively, adding a static version of encodeEvent to EventTranslator (decodeEvent is already static) would allow an EventRouter subclass (’RemoteRouter’) to call the serialization function without actually needing an EventTranslator instance.