The structure of the project/this presentation:

* Conducted a literature review into WSN protocols, platforms and the issues they face
* And then the project followed a generic implementation and testing lifecyle, but with more focus on the analysis
* We began by designing and implementing our own WSN protocol based on what we’ve already seen
* Optimised the network’s parameters such that it had very little idle time
* We produced response time models and evaluated them against the actual values
* Assesed how well the protocol handled external interference
* Conclusions and suggested further work from results

SLIDE

In brief, a WSN is a large interconnected network of resource constrained devices known as motes used for real-time environment montiring – they’re part of a real time system and increasingly used within applications from snail detection to more serious things like monitoring wildfire conditions

Protocol needs to automatically handle the effects of external interference (coming from outside the network) – whether that’s intentionally created to affect the network or someone putting their lunch in the microwave and it accidentally causing an effect. and also prevent internal interference i.e. make sure the transmissions don’t overlap with eachother

There are a wide range of protocols to choose from, but the common ground is the use of the IEEE 802.15.4 physical layer because of it slow rate low power capabilities. It provides two modes, either beacon enabled or non-beacon enabled with a large range frequencies to transmit on – known as channels.

SLIDE

We looked into two different protocols. The operation of NBE (e.g. zigbee) is essentially “transmit when you and the channel are ready”. BE – WirelessHART follows a completely deterministic synchornised ordering of transmission based on the assignment of timeslots (bandwidth) to transmissions which make up a superframe. This scheduling gets rid of the issue of overlapping transmissions as the devices know when they are allowed to transmit. External int is managed by channel hopping – changing channel each transmission and also blacklisitng certain frequencies when they are too noisy.

SLIDE

We also considered two execution platforms that ease development. TinyOS is an OS which uses a C-like language nesC. Mote Runner which supports most strictly typed HLLs and alo has a range of development tools – simulator means we don’t have to keep loading the physical motes. A brief look into response time analysis was also done to give us some background for when we produced our response time models

SLIDE

We began the project by defining a data transfer specification – just randomly generated that we wanted to implement with our new protocol. We assumed that firstly, all of the devices (A to E) were in the range of eachother such that they don’t have to route packets through the network and also that all the tarnsfers have the same period, such that none need to repeat before all have been completed.

SLIDE

Based on the speicifcation, our protocol was easy to base off of whart since it has deteminisim about when the transmissions will happen – zigbee is very random and can lead to starvation of some devices as others get all the bandwith. Hence we define our protocol, LikeWHART. Implemented in mote, we used the ame synchronisation policy with a centralized network manager. We employed channel hopping, but did this on a per superframe basis such that each time a superframe started a new channel was used. We were planning to implemented packet acknowledgement to ensure delivery of messages but Mote Runner lacked the documentation on the key functions with which to handle the packets and as such we couldn’t implement it.

SLIDE

So we began our definition of response time models based on two ordering policies – FPS (where we’ve already assigned priorities to the transfers in the speicication) and STF where we just order transmissions by their increasing size. Using our previous assumptions we produced two response time models that we’d later prove. We introduced the concept of the synchronisation constant, lamba which is the time between the end of the superframe and the start of the next one – effectively the idle time of transmissions – we’d investigate whether this is constant.

We have W being the size of the transfer, tau bein the maximum payload size of the network’s transmission and delta as the length of the timeslot.

SLIDE

For all of our analysis we employed the use of the simulator and use the logger system library ( a debugging tool) to act as packet acknolwegdment – when a device received a packet it outputted “packet received”. However these results need to be taken with a pinch of salt as the simulator is more of a software model of the motes rather than a hardware emulator – the actual physical motes may run differently.

We first began by optimizing the timeslot length (bandwith assigned) to devices to complete a transmission. Larger payload = more time so we optimized this for a range of payload sizes based on the spread of the data transfers specification – we wanted a 0 packet fault rate (number of packets lost) for the shortest timeslot possible.

TO find the synchronisation constant we just used the logger to measure the time between the end of a superframe and a device recivieng the synchronisation broadcast and we’d do this over all the optimized timlsots to ensure it is in fact constant.

We tested the intergirty of LikeWHART against external interference by introducing a rogue node that simply changed channel at a rate we’d increase in an attempt to see the effect on the packet fault rate

Finally we computed the theoretical response times for our transfer specification and then deployed it as an actual LikeWHART network and measured the real values to determine the accuracy of our model

Here’s a brief demo in the simulator….

SLIDE

So results.

We found that the optimal timeslot doesn’t actually double in size for a doubling of the payload size in the network it more slowly grows. We can see on the left however that the choice of timeslot is important as the packet fault rate grows exponentially when the timeslot gets increasingly too small for the transmission size.

We found the synchronisation constant was in fact constant at 25ms (on average) across all timeslot sizes.

We compared the response times to our theroretically computed response times for the data transfer specification and they were the same with only limited fluctuations but we put this down to the measuring apparatus.

Our integrity analysis was hardly unsurprisingly. LikeWHART doesn’t handle external interference at all well. As we increased the rate of channel hopping in the rouge device the packet fault rate grew rapidly since it has more chance of casuing a collision.

SLIDE

So what have we learnt?

* LikeWHART is a slow protocol – whart uses 10ms for its timeslot size and we got nowehere near that
* As mentioned, likewhart is terrible at handling external interference
* The synchronisation constant is a constant, but it’s also huge and needs to be minimized
* Our response time models were correct when compared to the actual values and STF has a much lower average response time for our data specification
* Essentially, LikeWHART needs a lot more work to be useful to anyone

SLIDE

So likewhart’s implementation was severely limited by the choice of mote runner especially packet acknowledgement. In the near furture we don’t have enough time to fix any of the funtiocnality so we’d just condut more analysis moving the physical motes to remove the doubt about the simulator. In the considerable future we’d change execution platform straight away and then be able to improve the integrirty handling of the protocol as well as make the topologies more complex and of course we’d define new response time models for these .