TYBSc CS 2020-2021 USCS607:Wireless Sensor Network

RollNo: 18043 Date: 24/02/2021

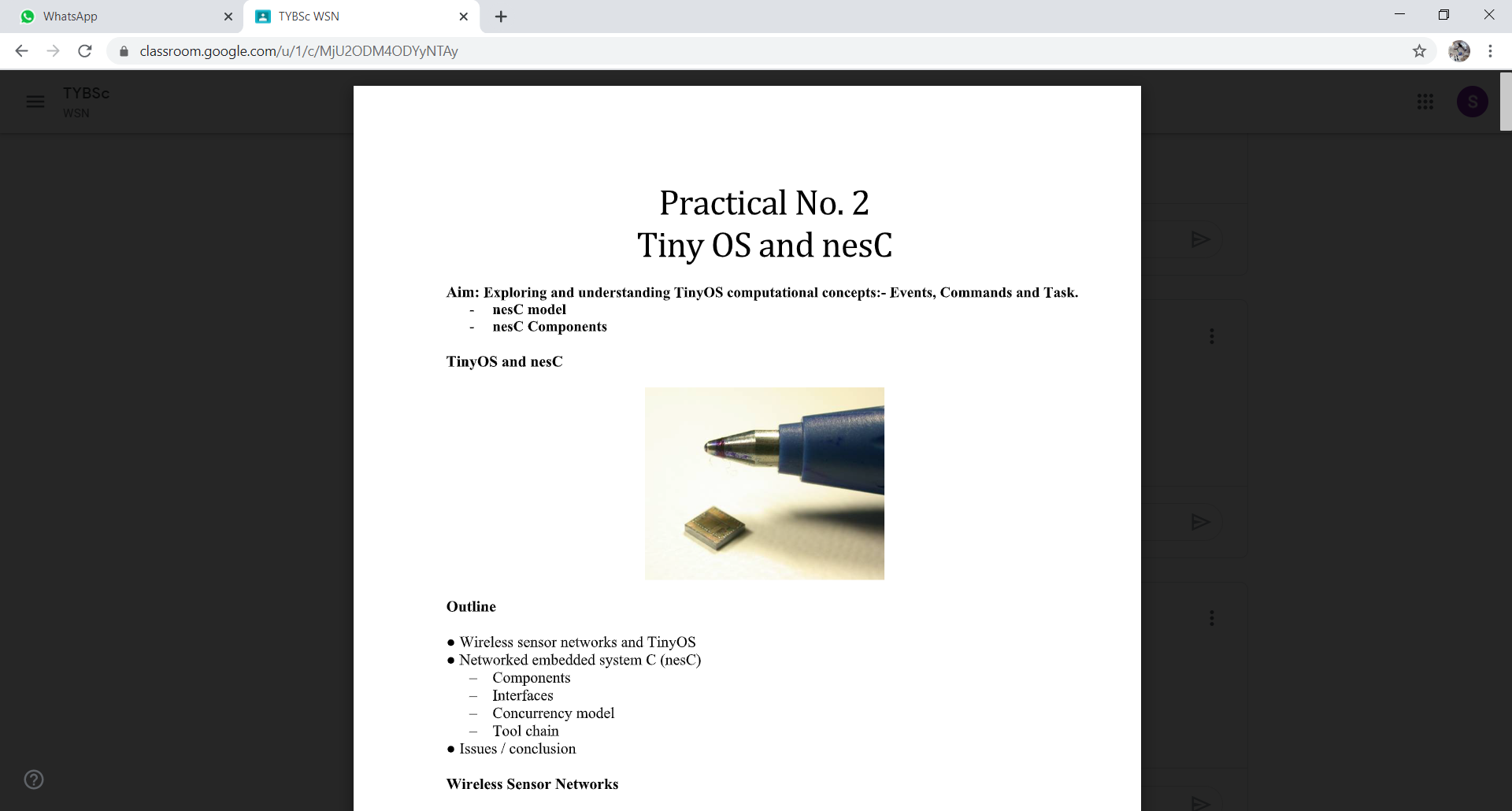
**Practical No. 2**

**Tiny OS and nesC**

**Aim:** Exploring and understanding TinyOS computational concepts:- Events, Commands and Task.

* nesC model
* nesC Components

**TinyOS and nesC**



**Outline**

* Wireless sensor networks and TinyOS
* Networked embedded system C (nesC)
  + Components
  + Interfaces
  + Concurrency model
  + Tool chain
* Issues / conclusion

**Wireless Sensor Networks:-**

* Vision: ubiquitous computing
* Extreme dynamics
* Interact with environment using sensors and radio
* Immense scale
* Limited access
* Small, cheap, low-power systems

**Concepts in Sensor Networks:-**

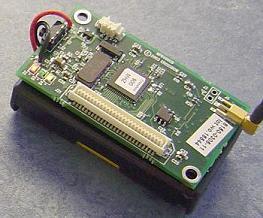
* In-network processing and data aggregation
* Radio activity 1000 times as expensive as processing
* Duty-cycling: different modes of operation
* Power down unused hardware
* Systems run a single application
* Applications are deeply tied to hardware
* Require customized and optimized OS

**Challenges:-**

* Limited resources: energy consumption dominates
* Concurrency: driven by interaction with environment
* Soft real-time requirements
* Reliability: reduce run-time errors, e.g. races
* High diversity of platforms
* No well-defined software/hardware boundary

**TinyOS:-**

* Component-based architecture
* Reusable system components: ADC, Timer, Radio
* Tasks and event-based concurrency
* No user-space or context switching supported by hardware
* Tasks run to completion only preempted by interrupts
* All long-latency operations are split-phase
  + Operation request and completion are separate functions



**Introducing nesC:-**

* A “holistic” approach to networked embedded systems
* Supports and reflects TinyOS&#39;s design
* Extends a subset of C
* A static language
  + All resources known at compile-time
  + Call-graph fully known at compile-time

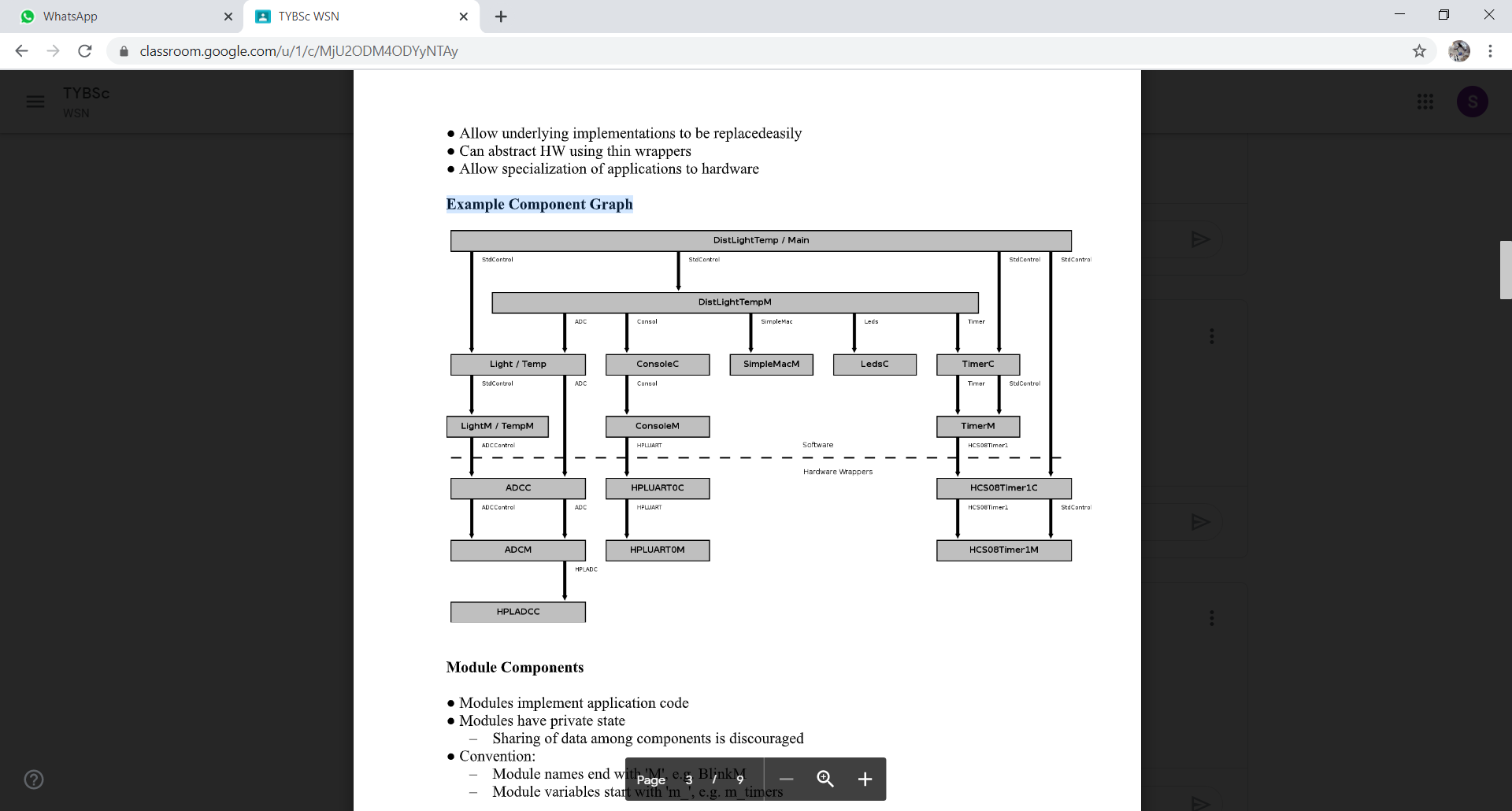
**Design Decisions for nesC:-**

* Components
* Bidirectional interfaces
* Simple expressive concurrency model
* Whole-program analysis

**Components:-**

* Challenge: platform diversity, flexible SW/HW boundary, applications deeply tied to hardware
* Encourages modular design
* Restrict access to private data
* Allow underlying implementations to be replaced easily
* Can abstract HW using thin wrappers
* Allow specialization of applications to hardware

**Example Component Graph:-**



**Module Components:-**

* Modules implement application code
* Modules have private state
  + Sharing of data among components is discouraged
* Convention:
  + Module names end with &#39;M&#39;, e.g. BlinkM
  + Module variables start with &#39;m\_&#39;, e.g. m\_timers

**Configuration Components:-**

* Configurations wire other components together
* All applications have a top-level configuration
* A component interface may be wired zero or moretimes
  + Used for StdControl to implement power management
* Convention:
* Configuration names end with &#39;C&#39;, e.g. TimerC

(unless it is the top-level configuration ;-)

**Modules and Configurations:-**

**/\* BlinkM.nc \*/**

**module** BlinkM {

**provides interface** StdControl as Control;

**uses interface** Timer;

**uses interface** Leds;

} **implementation** {

**command** result\_t Control.init() {

**call** Leds.init();

**return** SUCCESS;

}

**command** result\_t Control.start() { /\* ... \*/ }

**command** result\_t Control.stop() { /\* ... \*/ }

**event** result\_t Timer.fired() {

**call** Leds.redToggle();

**return** SUCCESS;

}

}

**/\* Blink.nc \*/**

**configuration** Blink {

} **implementation** {

/\* Declare used components. \*/

**components** Main, BlinkM, SingleTimer, LedsC;

/\* Wire components together. \*/

Main.StdControl -&gt; SingleTimer.StdControl;

Main.StdControl -&gt; BlinkM.StdControl;

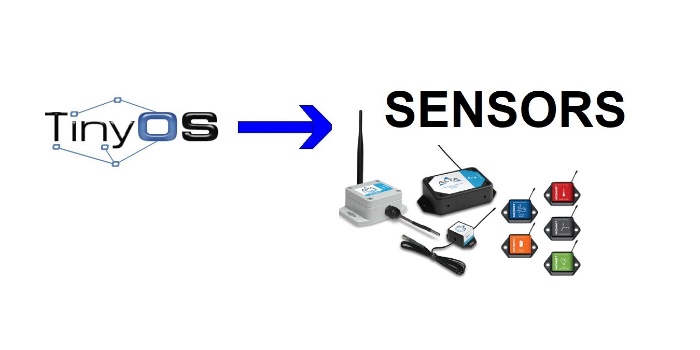
BlinkM.Timer -&gt; SingleTimer.Timer;

BlinkM.Leds -&gt; LedsC;

}

**Bidirectional Interfaces:-**

* Challenge: flexible SW/HW boundary andconcurrency
* Support split-phase operations
* Commands: call down the component graph
  + Implemented by provider
* Events: call up the component graph
  + Implemented by user



**Interfaces:-**

**/\* Timer.nc \*/**

**includes** Timer; /\* Include C types from Timer.h \*/

**interface** Timer {

**command** result\_t start(char type, uint32\_t interval);

**command** result\_t stop();

**event** result\_t fired();

}

**/\* SyncAlarm.nc \*/**

**interface** SyncAlarm&lt;Precision\_t&gt; {

**command** result\_t armCountdown(Precision\_t timeout);

**command** result\_t armAlarmClock(Precision\_t time);

**command** result\_t stop();

**event** result\_t alarm();

}

**Parameterized Interfaces:-**

**module** TimerM {

**provides** interface Timer[uint8\_t id];

} **implementation** {

/\* ... \*/

Timer\_t m\_timers[NUM\_TIMERS];

**command** result\_t Timer.isSet[uint8\_t timer]() {

return m\_timers[timer].isset;

}

**task** void timerCheck() {

uint8\_t timer;

for (timer = 0; timer &lt; NUM\_TIMERS; timer++)

if (m\_timers[timer].fired)

**signal** Timer.fired[timer]();

}

/\* ... \*/

}

**configuration** MyApp { /\* ... \*/ }

**implementation** {

**components** MyAppM, TimerC, /\* ... \*/;

MyAppM.SampleTimer -&gt; TimerC.Timer[unique(“Timer”)];

}

**Concurrency Model:-**

* Challenge: extreme dynamics and soft real-timerequirements
* Cooperative scheduling
* Light-weight tasks
* Split-phase operations: non-blocking requests
* Built-in atomic sections
  + Limited crossing of module boundaries

**Sources of Concurrency:-**

* Tasks
  + Deferred computation
  + Run sequential and to completion
  + Do not preempt
* Events
  + Run to completion, and may preempt tasks and events
  + Origin: hardware interrupts or split-phase completion

**Tasks and Events:-**

**module** LightM {

/\* ... \*/

} **implementation** {

uint16\_t light\_data

**task** void processLightdata() {

uint16\_t local\_light\_data;

**atomic** local\_light\_data = light\_data;

/\* Process light data. \*/

**if** (!done)

**post** anotherTask()

}

**async** event result\_t Light.dataReady(uint16\_t data) {

**atomic** lightData = data;

**post** processLightData();

**return** SUCCESS;

}

**event** result\_t SensorTimer.fired() {

**return** call Light.getData();

}

}

**Whole-Program Analysis:-**

* Compilation can examine complete call-graph
  + Remove dead-code
  + Eliminate costly module boundary crossings
  + Inline small functions
* Back-end C compiler can optimize whole program
  + Perform cross component optimizations
  + Constant propagation, common sub-expression elimination
* Allows detection of race conditions

**Synchronous and Asynchronous:-**

* Asynchronous code (AC):
  + Code reachable from at least one interrupt handler
  + Events signaled directly or indirectly by hardware interrupts
* Synchronous code (SC):
  + “Everything else ...”
  + Primarily tasks

**Detecting Race Conditions:-**

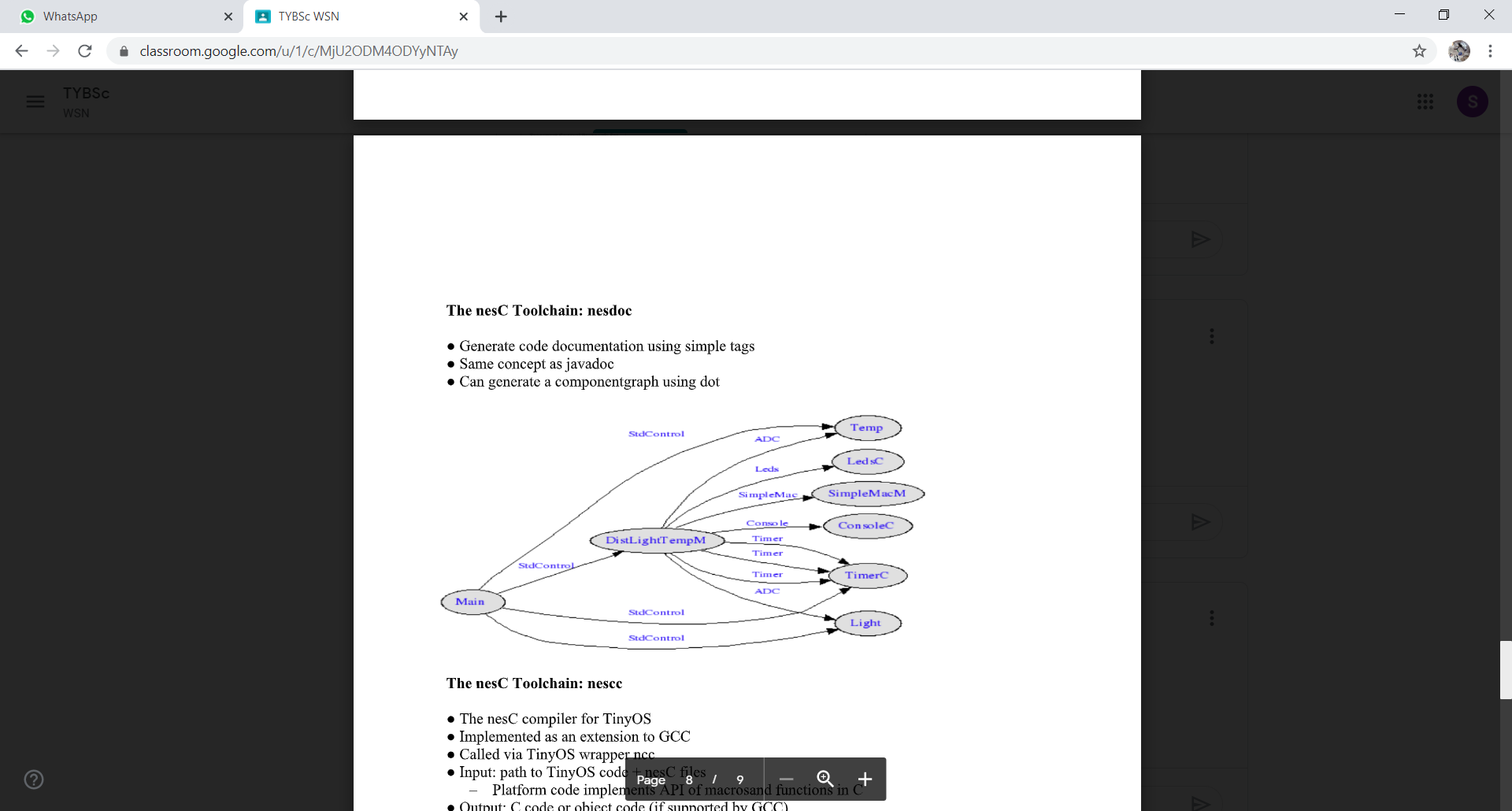
* Invariant: SC is atomic with respect to other SC
* Two claims about updates for AC/AC and SC/AC:
  + Any update to shared state from AC is a potential race condition
  + Any update to shared state from SC that is also updated from AC is a potential race condition
* Race-free invariant enforced at compile time:
  + Updates to shared state is either SC only or in atomic section

**Dealing with Race Conditions:-**

* Use atomic sections to update shared state
  + atomic { shared\_var = 1; }
* Convert code with updates to shared state to tasks
* Mark false positive with norace qualifier
  + norace uint8\_t variable;

**The nesC Toolchain: nesdoc**

* Generate code documentation using simple tags
* Same concept as javadoc
* Can generate a componentgraph using dot



**The nesC Toolchain: nescc:-**

* The nesC compiler for TinyOS
* Implemented as an extension to GCC
* Called via TinyOS wrapper ncc
* Input: path to TinyOS code + nesC files
  + Platform code implements API of macrosand functions in C
* Output: C code or object code (if supported by GCC)

**The nesC Toolchain: ncg and mig:-**

* Allows integration with Java code
* Typical use: interact with network through basestation
* ncg - extract constants from nesC files
  + Generates class that contains constants
* mig - message interface generator for nesC
  + Generates class that encodes and decodes messages

**Issues for nesC:-**

* Problem for data shared across components
  + False positives for buffer swapping
* Problem for data shared between split-phase operations
  + Event can potentially fire if other components access HW
* Some TinyOS idioms are not well expressed
  + Parameterized interfaces each with private state

**Issues for Applications:-**

* Focus early on modeling it as a state-machine
* Design duty-cycling from the start
  + Affect the state-machine so hard to add later
* Abstracting functionality into components
  + Makes it harder to access shared state:
* encapsulate shared state in a separate module
* Configuring TinyOS for the application needs
  + By default there can only be 8 posted tasks

**Conclusions for nesC:-**

* Bidirectional interfaces fit the TinyOS model
* Components are a good abstraction
* Concurrency model meets requirements inapplications
* The restrictions in nesC introduce practical problems
* Not limited to the domain of embedded systems

