CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

utc Université de Technologie Compiègne

TinyOS: Système d'exploitation pour réseaux de capteurs sans fils

Y. Challal

Contributeurs

- **Hatem Bettahar**
- **Boushra Maala**
- **Abdelraouf Ouadjaout**
- **Noureddine Lasla**
- **Mouloud Bagaa**
- **Ben Hamida Fatima Zohra**

- **Xufei Mao**
- **Chenyang Lu (Virginia)**
- **Kemal Akkaya and Mohamed Younis**
- **C. Intanagonwiwat, R. Govindan, D. Estrin, et. al., presented by Romit Roy Choudhury (Illinois)**
- **Martin Haenggi, « Wireless Sensor Networks »**

INTRODUCTION

Système d'exploitation Classqiue

- **Enorme !**
- **Architecture Multi-thread=>Mémoire importante**
- **Modèle E/S**
- **Séparation entre espace noyau et utilisateur**
- **Pas de contraintes d'énérgie**
- **Ressources disponibles**

Contraintes matérielles d'un "Mote"

- **Ressources énérgétiques basses**
- **Mémoire limitée**
- **CPU lente**
- **Petite taille**
- **Parallélisme matériel limité**
- **Communication radio**
  - Bande-passante faible
  - Portée radio courte

Propriétés du SE désiré

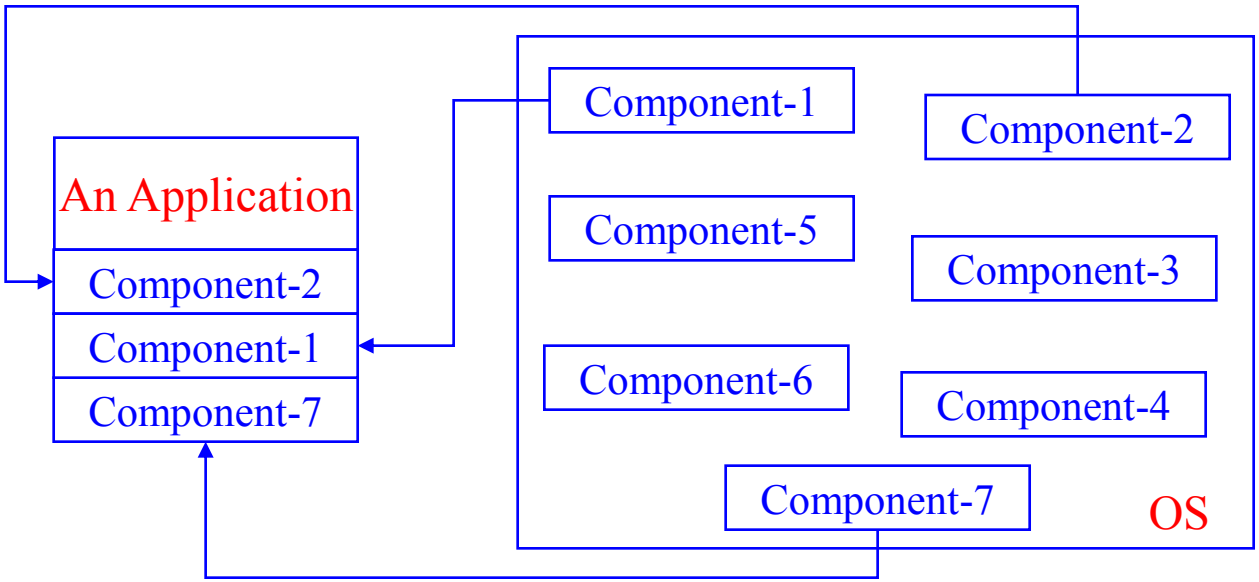
- **Image mémoire petite**
- **Efficacité en calcul et consommation d'énérgie**
- **La communication est fondamentale**
- **Temps-réel**
- **Construction efficace d'applications**

La solution TinyOS

- **Concurrence : utilise une architecture orientée événement**
- **Modularité**
  - Application composée de components
  - SE + Application compilés en un seul exécutable
- **Communication**
  - Utilise un modèle event/command
  - Ordonnancement FIFO non préemptif
- **Pas de séparation kernel/application**

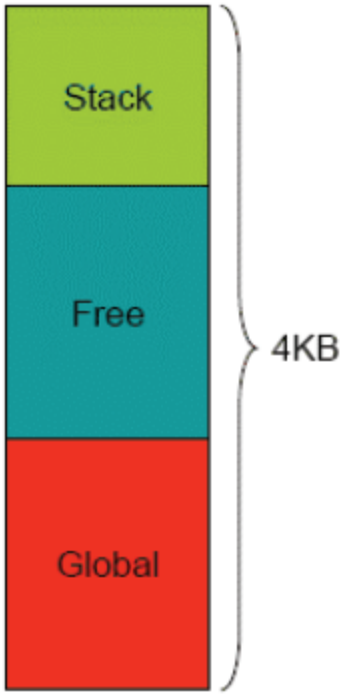
Aperçus générale de TinyOS

- **Système d'exploitation pour réseaux de capteurs embarqués**
- **Ensemble de composants logiciels qui peuvent être reliés ensemble en un seul exécutable sur un "mote"**
- **Fonctions minimales**
  - 2 threads: tâches et "handlers" d'événements matériels
  - Pas de gestion de la mémoire…



Modèle mémoire de TinyOS

- **Allocation statique de la mémoire**
  - Pas de heap (malloc)
  - Pas de pointeur sur function
  - Pas d'allocation dynamique
- **Variables globales**
  - Disponibles per-frame
  - Conservation de la mémoire
  - Utilisation de pointeurs
- **Variables locales**
  - Sauvegardées sur la pile (stack)
  - Déclarées dans une méthode



Le langage de programmation de TinyOS

NESC

nesC: Aperçus

- **Prononcé ”NES-see”**
- **Extension du langage C**
  - Supporte la syntaxe C
  - Compilé vers C
- **“conçus pour incarner les concepts structurant et le modèle d’exécution de TinyOS”**
- **TinyOS est originalement écrit en C, les applications étaient une combinaison de fichiers .c, .comp, et .desc**
- **Les composants de TinyOS ont été réimplémentés en nesC**

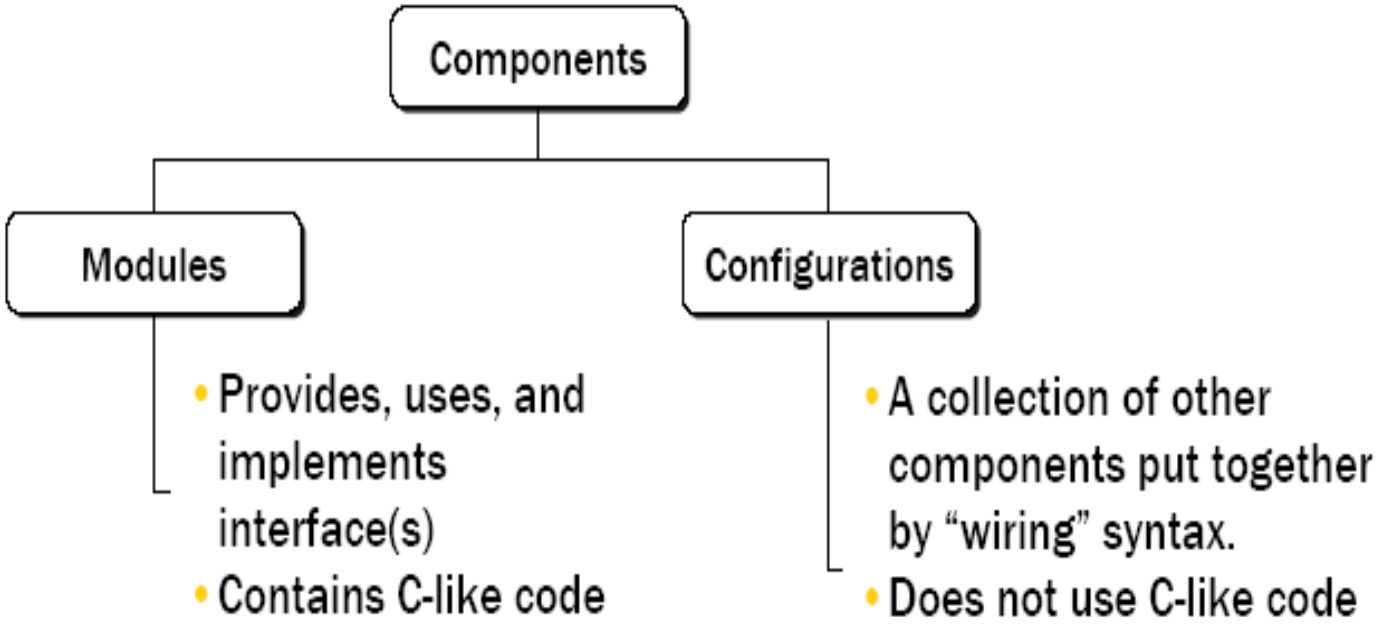
Concepts de nesC

- **Unité de code de base = Composant**
- **Composant**
  - Exécute des Commands
  - Lance des Events
  - Dispose d'un Frame pour stocker l'état local
  - Utilise la notion de Tasks pour gérer la concurence
- **Un Composant implémente des interfaces**
  - Utilisées par d'autres composants pour communiquer avec ce composant

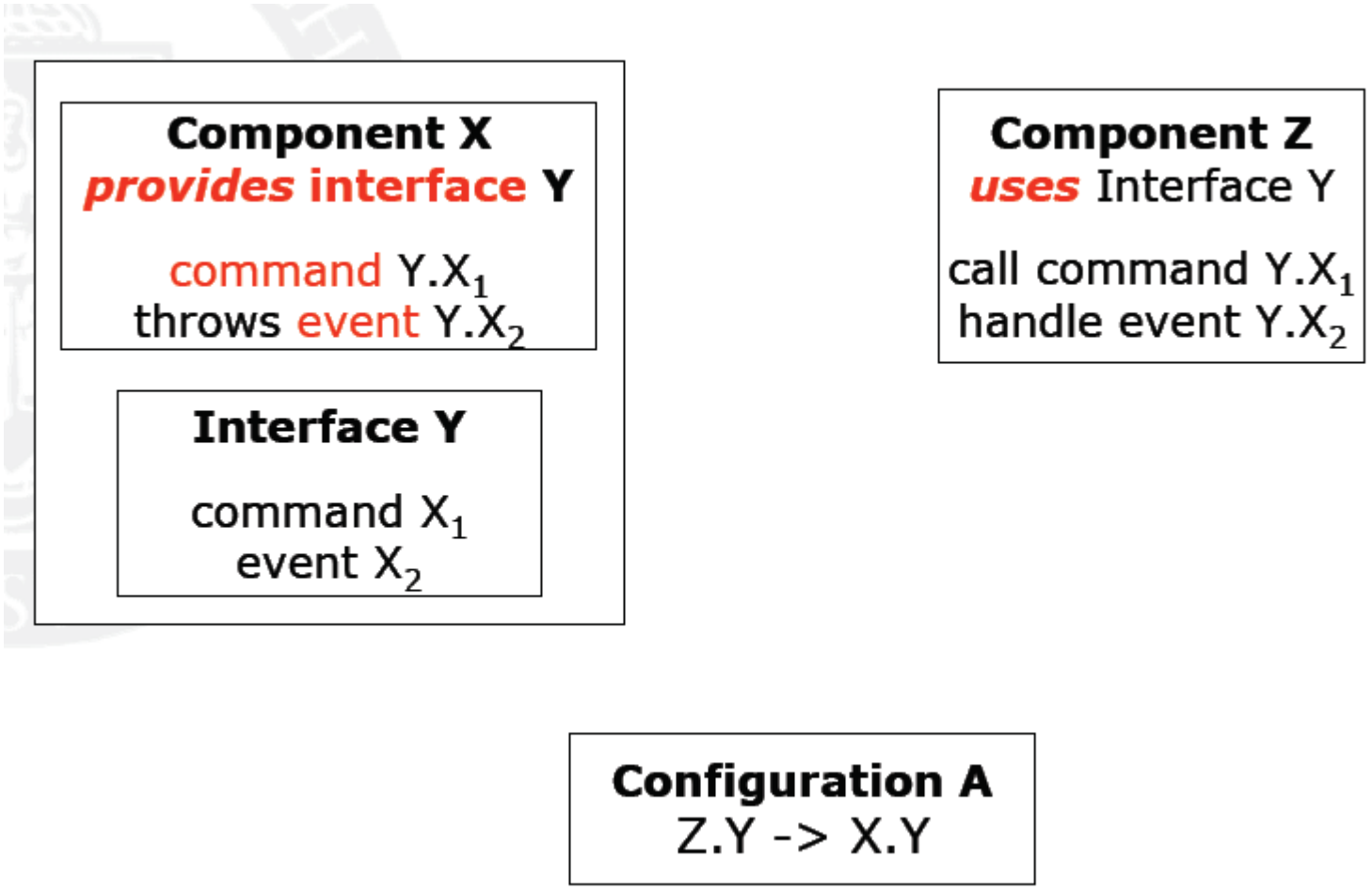
Composants

- **Deux types de composants**
  - Module
    - ✓ composant implémenté avec du code
  - Configuration
    - ✓ Composants reliés ensemble pour former un autre composant

Composants


Components
Modules
Configurations
• Provides, uses, and implements interface(s)
• Contains C-like code
• A collection of other components put together by "wiring" syntax.
• Does not use C-like code

Application TinyOS (Exemple)



Résumé vocabulaire

- **Application – un ou plusieurs composants reliés ensemble pour former un exécutable**
- **Composant – un élément de base pour former une application nesC. Deux types: modules et configurations**
- **Module – composant qui implémente une ou plusieurs interfaces**
- **Configuration – composant qui relie d'autres composant ensemble**
- **Interface – définie d'une manière abstraite les interactions entre deux composants**

Interfaces

- **Une interface définie les interactions entre deux composants.**
- **Deux types d'opérations: commands, events**

```
interface StdControl
{
    command result_t init();
    command result_t start();
    command result_t stop();
}
```

```
interface X
{
    command result_t doSomething();
    event result_t doSomethingDone();
}
```

Interfaces, commands, events

- **Les interfaces sont bidirectionnelles: “Elles spécifient un ensemble de fonctions à implémenter par les composants fournisseurs de l’interface (commands), et un ensemble à implémenter par les composants utilisateurs de l’interface (events)”**

- **Les “commands” font typiquement des appels du haut vers le bas (des composants applicatifs vers les composants plus proches du matériel), alors que les “events” remontent les signaux du bas vers le haut.**

Exemple d'une interface

```
// can include c files


interface SendMsg {
command result_t send(uint16_t address, uint8_t length,
    TOS_MsgPtr msg);
event result_t sendDone(TOS_MsgPtr msg, result_t
    success);
}
```

Appeler une command et signaler un event

- **Appeler une “command”**

```
call Send.send(1, sizeof(Message), &msg1);
```

- **Signaler un “event”**

```
signal Send.sendDone(&msg1, SUCCESS);
```

Modules

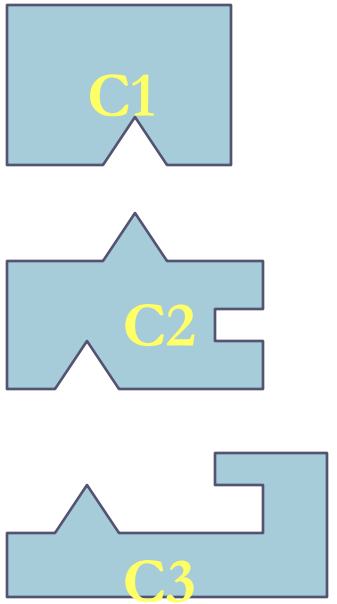
- **Implemente une ou plusieurs interfaces**
- **Peut utiliser une ou plusieurs interfaces**

```
module Provider
{
   provides interface StdControl;
   provides interface X;
   uses interface Z;
}
implementation
{
    // C code
    ....
}
```

Modules (suite)

- **modules:**

```
module C1 {
  requires interface triangle;
} implementation { ... }
module C2 {
  provides interface triangle in;
  requires {
    interface triangle out;
    interface rectangle side; }
} implementation { ... }
module C3 {
  provides interface triangle;
  provides interface rectangle;
} implementation { ... }
```



Modules (Exemple)

Implementor

```
module Provider
{ provides interface StdControl;
  provides interface X;
  uses interface Z;
}
implementation {
  command result_t StdControl.init()
  { return SUCCESS; }

  command result_t StdControl.start()
  { return SUCCESS; }

  command result_t StdControl.stop()
  { return SUCCESS; }

  task void signaler()
  {
    signal X.doSomethingDone();
  }

  command result_t X.doSomething()
  {
     post signaler();
     return SUCCESS;
  }}
```

User Component

```
module User
{ ...
  uses interface X;
}
implementation {

 ....

  task void A()
  {
    res = call X.doSomething();
  }

 .....

 event result_t X.doSomethingDone()
 {
     // Yay ! doSomething returned ok
     return SUCCESS;
 }
}
```

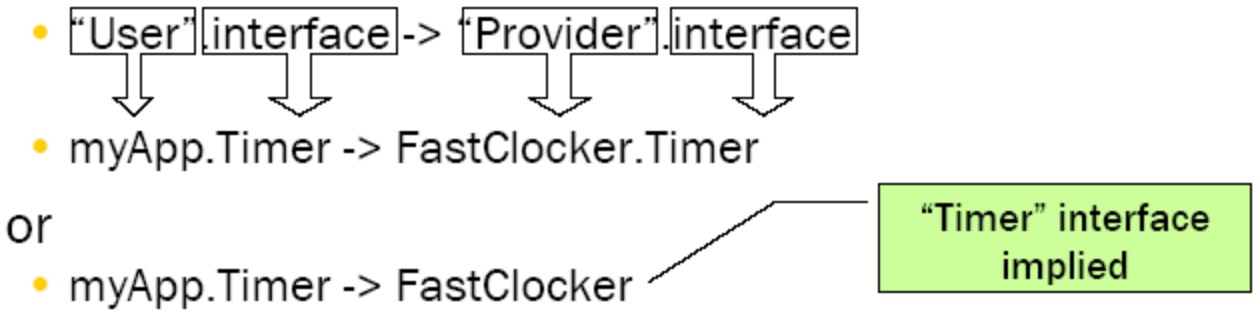
Configurations

- **Dans nesC, deux composants sont reliés ensemble en les connectant (wiring)**
- **Les interfaces du composant utilisateur sont reliées (wired) aux mêmes interfaces du composant fournisseur**
- **3 possibilités de connexion (wiring statements) dans nesC:**
  - endpoint1 = endpoint2
  - endpoint1 -> endpoint2
  - endpoint1 <- endpoint2 (equivalent: endpoint2 -> endpoint1)

nesC Wiring Syntax

- **Les éléments connectés doivent être compatibles**
  - Interface-interface, commande-command, event-event
    - ✓ On peut connecter Send à Send
    - ✓ Mais pas Send à receive (par exemple)

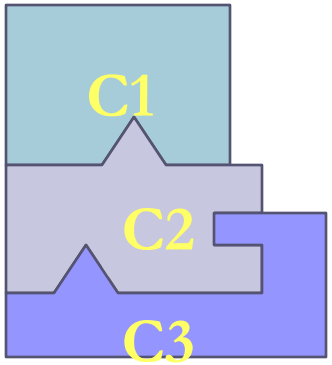
- **Connecter un utilisateur d'une interface à un composant qui implémente l'interface:**



Configurations (illustration)

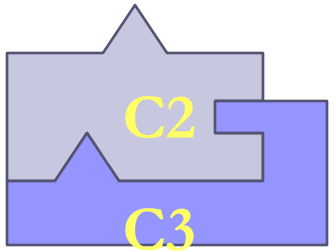
Connecter des configurations:

```
configuration app { }
implementation {
    uses c1, c2, c3;
    c1 -> c2; // implicit interface sel.
    c2.out -> c3.triangle;
    c3 <- c2.side;
}
```



Configurations partielles:

```
component c2c3 {
  provides interface triangle t1;
}
implementation {
    uses c2, c3;
    t1 -> c2.in;
    c2.out -> c3.triangle;
    c3 <- c2.side;
}
```



Configurations (Exemple)

Application.nc

```
configuration Application {
}
implementation {
  components Main, Provider, User, SomeComp;

  Main.StdControl -> Provider.StdControl;
  User.X -> Provider.X;
  Provider.Z -> SomeComp.Z;
}
```

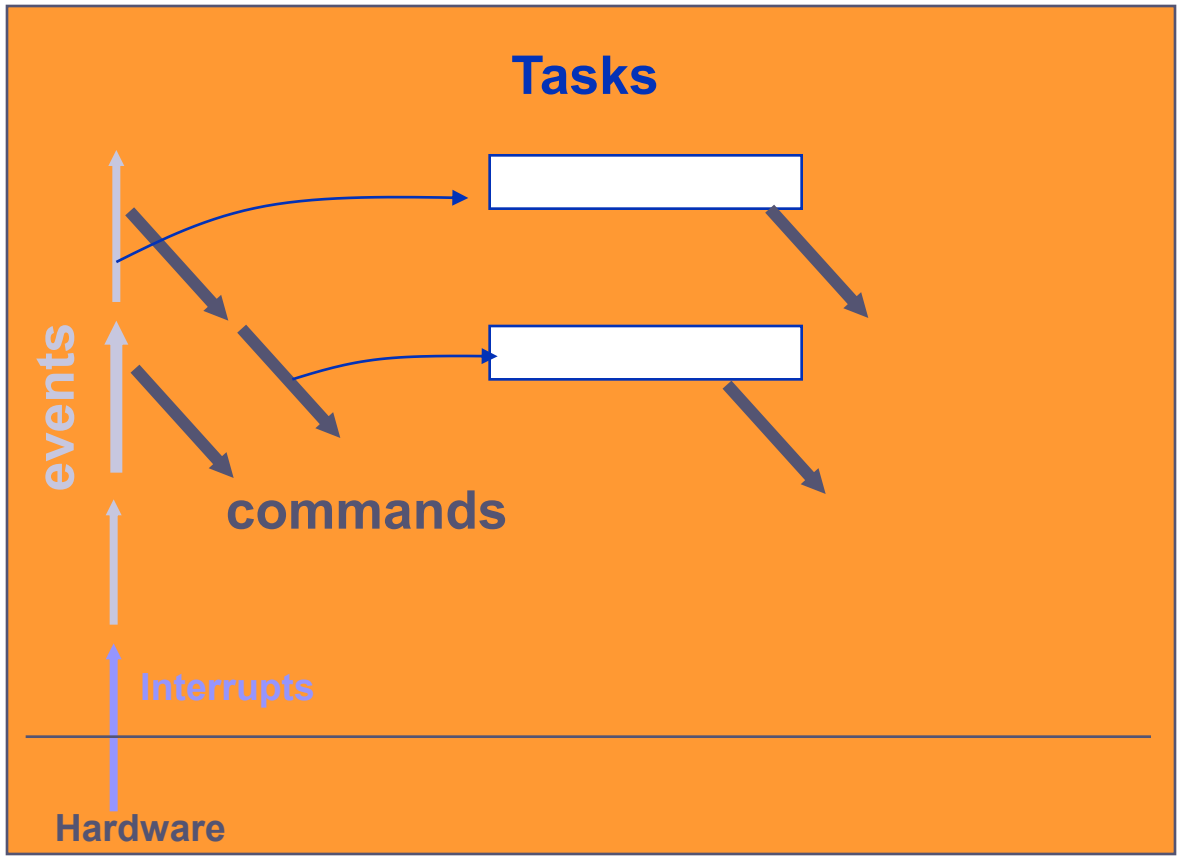
Tasks

- **Une tâche est un élément de contrôle indépendant défini par une fonction retournant void et sans arguments:**
  - task void myTask() { ... }”
- **Les tâches sont lancées en les préfixant par “post“**
  - post myTask();”

Commands/Events/Tasks

- **Command**
  - Ne doit pas être bloquante i.e. prend les paramètres, commence le traitement et retourne dans l'application;
  - Reporte le travail qui consomme du temps en postant une tâche
  - Peut appeler des commandes sur d'autres composants
- **Event**
  - Peut appeler des commandes, signaler d'autres événements, poster des tâches mais ne peut pas être signalé par des commandes
  - Peut interrompre une tâche mais pas l'inverse.
- **Task**
  - Ordonnancement FIFO
  - Non préemptive par une autre tâche, préemptive par un événement
  - Utilisée pour réaliser un travail qui nécessite beaucoup de calculs
  - Peut être postée par une commande ou un event.

TinyOS Execution Contexts



Instructions atomiques

- **Guarantie que l'exécution de l'instruction se fait comme si aucun autre calcul ne se fait simultanément**
- **Doit être courte**
- **nesC interdit dans une instruction atomique: call, signal, goto, return, break, continue, case, default, label**

```
bool busy; // global
void f() {
  bool available;
  atomic {
    available = !busy;
    busy = TRUE;
  }
  if (available) do_something;
  atomic busy = FALSE;
}
```

Conventions de nommage dans nesC

- **Extension des fichiers nesC: .nc**
- **Clock.nc : soit une interface (ou une configuration)**
- **ClockC.nc : une configuration**
- **ClockM.nc : un module**

nesC Keywords

interface	A collection of event and command definitions
module	A basic component implemented in nesC
configuration	A component made from wiring other components

implementation	Contains code & variables for module or configuration
components	List of components wired into a configuration

provides	Defines interfaces provided by a component
uses	Defines interfaces used by a module or configuration
as	Alias an interface to another name

command	Direct function call exposed by an interface
event	Callback message exposed by an interface

nesC Keywords -implementation

call	Execute a command
signal	Execute an event
post	Put a task on the execution queue
task	A function to be executed in the background

includes	Include a header file

async	For commands, events executed asynchronously
atomic	Guarantees atomic execution of a statement
norace	Eliminates warnings of race conditions nesC detected

Compilation et excéution

- **Le compilateur traîte les fichiers nesC en les convertissant en un fichier C "gigantesque"**
  - Il contient votre application et les composants du SE utilisés par votre application
- **Ensuite un compilateur spécifique à la plateforme cible compile ce fichier C**
  - Il devient un seul exécutable
- **Le chargeur installe le code sur le Mote (Mica2, Telos, etc.)**

Exemple

APPLICATION BLINK

Exemple Blink

- **Un programme qui allume la LED rouge chaque second**

Blink.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

BlinkM.nc (specification)

```
module BlinkM {
  provides {
    interface StdControl;
  }  uses {
    interface Timer;
    interface Leds;
  }
}
```

BlinkM.nc (implementation)

```
implementation {

  command result_t StdControl.init() {
    call Leds.init();
     return SUCCESS;
  }

  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired()  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```