

Καταμεμημένα Συστήματα I

Μάθημα Βασικής Επιλογής,

Χειμερινού Εξαμήνου

Τομέας Εφαρμογών και Θεμελιώσεων

Χρήστος Κονίνης

Τρίτη, 24 Νοεμβρίου, 2009
Υπολογιστικό



Επισκόπηση

Ανασκόπηση

Εκτέλεση μιας εξομοίωσης
Πώς φτιάχνουμε μια νέα εφαρμογή στο shawn
The Processor
The Messages

JSawn

JSawn

Ασκήσεις

Ασκηση Μερos Β



Εκτέλεση μιας εξομοίωσης

- ▶ Για να εκτελέσουμε μια εξομοίωση πρέπει να δώσουμε στο shawn ένα σύνολο από εντολές

- ▶ Τις εντολές αυτές συνήθως τις γράφουμε σε ένα αρχείο με κατάληξη .conf (π.χ. hello.conf) στο φάκελο shawn/buildfiles

Παράδειγμα ενός hello.conf

```
prepare_world edge_model=simple comm_model=disk_graph range=2  
rect_world width=5 height=5 count=30 processors=helloworld  
simulation max_iterations=10
```

- ▶ Το παραπάνω hello.conf λέει στο shawn
 1. Να κατασκευάσει ένα νέο περιβάλλον εξομοίωσης (prepare_world)
 2. Να γεμίζει το περιβάλλον 5x5 με 20 κόμβους που να θα τρέχουν την εφαρμογή helloworld (rect_world)
 3. Και τέλος να εκτελέσει την εξομοίωση για 10 γύρους (simulation)
- ▶ Τέλος τρέχουμε το shawn με την εντολή: ./shawn -f hello.conf



Πώς εισάγουμε ένα template application στο shawn

- ▶ Κατεβάζουμε το φάκελο του template που περιέχει όλα τα απαραίτητα αρχεία που υλοποιούν ένα καταμεμημένο αλγόριθμο π.χ. :

```
http://www.ceid.upatras.gr/courses/katanemihmena/  
wiki/Images/0/00/SHAWN-FloodMax-Template.tar.bz2
```

- ▶ Αντιγράφουμε τον φάκελο στην θέση shawn/src/legacypapps/
- ▶ Από την κονσόλα ηγαινούμε στον φάκελο shawn/buildfiles
- ▶ Δίνουμε την εντολή cmake ../src και μετά:
 - ▶ c , για να ενημερωθεί για τον νέο κώδικα που προσθέσατε
 - ▶ Παρατηρούμε πως έχει εμφανιστεί μια νέα επιλογή στην πρώτη θέση με το όνομα της εφαρμογής
 - ▶ Ενεργοποιήστε την επιλογή (πατώντας enter)
 - ▶ c , για να ενημερωθεί για τις αλλαγές
 - ▶ g , για να αποθηκεύσει τις αλλαγές
- ▶ Την παραπάνω διαδικασία την εκτελούμε **μόνο** όταν προσθέτουμε ένα νέο αλγόριθμο στον φάκελο legacypapps
- ▶ Τέλος δώστε την εντολή make για να γίνει compile to shawn



Πώς φτιάχνουμε ένα template application (1 από 2)

1. Κατασκευάζουμε ένα νέο φάκελο μέσα στο `/shawn/src/legacyapps`, π.χ. με το όνομα `simp_app`
2. Αντιγράφουμε όλα τα αρχεία `helloworld_processor*` και `helloworld_message*`, `module.cmake` και `examples_init*` από τον φάκελο `src/apps/examples/processor` στο `/shawn/src/legacyapps/simp_app/`
3. Τελικά ο φάκελος θα περιέχει τα παρακάτω αρχεία :
 - ▶ `module.cmake`
 - ▶ `simple_app_init.h`
 - ▶ `simple_app_init.cpp`
 - ▶ `simple_app_message.h`
 - ▶ `simple_app_message.cpp`
 - ▶ `simple_app_processor.h`
 - ▶ `simple_app_processor.cpp`
 - ▶ `simple_app_processor_factory.h`
 - ▶ `simple_app_processor_factory.cpp`



Πώς φτιάχνουμε ένα template application (2 από 2)

1. Αφού μεταφέρουμε και μετονομάσουμε τα αρχεία πρέπει να τροποποιήσουμε τον κώδικα
2. Αντικαθιστούμε κάθε εμφάνιση του `helloworld` στα αρχεία με `simple_app` (και `HelloWorld` με `SimpleApp` αντίστοιχα)
3. Στο αρχείο `module.cmake` τροποποιούμε μόνο μια γραμμή:

```
set ( moduleName SIMPLE_APP )
```
4. Ενεργοποιούμε και κάνουμε `compile` τον νέο κώδικα ακολουθώντας την διαδικασία που περιγράψαμε παραπάνω
5. Τώρα μπορούμε να χρησιμοποιήσουμε τα αρχεία για να γράψουμε ένα νέο καταναμημένο αλγόριθμο



Τα αρχεία μιας εφαρμογής

- ▶ **module.cmake :**
Περιέχει το όνομα της νέας εφαρμογής που ενεργοποιούμε με το `cmake`
- ▶ **simple_app_message:**
Περιέχει τα μηνύματα που στέλνει ο αλγόριθμος
- ▶ **simple_app_processor:**
Περιέχει τον κώδικα και τις μεταβλητές του αλγορίθμου, εδώ υλοποιούμε όλη την λογική της εφαρμογής μας
- ▶ **simple_app_processor_factory:**
Παρέχει 2 σημαντικές συναρτήσεις, την `create()` και `name()`, που φτιάχνει ένα νέο `processor` και επιστρέφει το όνομά του
- ▶ **simple_app_init :**
Προσθέτει το `simple_app_processor_factory` στον εξομοιωτή καλωπας την `register_factory`, ώστε το `shawn` να μπορεί να φτιάξει νέα αντικείμενα του `processor`



The Processors

- ▶ Μας επιτρέπουν να υλοποιούμε την λογική των καταναμημένων αλγορίθμων μας
- ▶ Μπορούμε να έχουμε πολλούς `processors` σε ένα κόμβο
 - ▶ Αλγόριθμο `Tree` Ρουτιν
 - ▶ Αλγόριθμο `Leader` ελεςπιον
- ▶ Έχουν τρεις καταστάσεις
 - ▶ `Active`: Πλήρη λειτουργία
 - ▶ `Sleep`: Δεν δέχονται μηνύματα
 - ▶ `Inactive`: Καμία λειτουργία
- ▶ Αν όλοι οι `processors` σε ένα κόμβο είναι `Inactive` τότε ο κόμβος γίνεται `Inactive`
- ▶ Αν όλοι οι κόμβοι είναι `Inactive` η εξομοίωση τερματίζει



Συναρτήσεις που μπορείτε να υλοποιήσετε (1 of 2)

Ακολουθεί μια λίστα με βασικές συναρτήσεις που μπορείτε να **υλοποιήσετε** μέσα στον processor

▶ void boot()

Εκτελείται μια φορά για κάθε processors στην αρχή της εξομοίωσης. Χρησιμοποιείται συνήθως για αρχικοποίηση του αλγορίθμου μας.

▶ void special_boot()

Εκτελείται μια φορά για κάθε processors που βρίσκεται σε ένα ειδικό (special) κόμβο. Καλείται πριν την boot()

▶ void work()

Εκτελείται μια φορά για κάθε γύρο (βήμα) της εξομοίωσης. Χρησιμοποιείται για την εκτέλεση περιοδικών tasks.



Συναρτήσεις που μπορείτε να υλοποιήσετε (2 of 2)

▶ bool process_message(MessageHandle&)

Εκτελείται όταν ο κόμβος λάβει ένα μήνυμα. Το μήνυμα παραδίδεται στους processors του κόμβου. Μπορεί να λάβει διαφορετικά μηνύματα οπότε πρέπει να εξετάζουμε τον τύπο του μηνύματος.

```
bool
MyProcessor::
process_message( const shawn::ConstMessageHandle& mh ){
    const MyMessage* mymsg =
        dynamic_cast<const MyMessage*>( mh.get() );

    if ( MyMessage!= NULL ) {
        // handle my message
        return true;
    }

    return shawn::Processor::process_message( mh );
}
```



Χρήσιμες συναρτήσεις που μπορείτε να καλέσετε

Ακολουθεί μια λίστα με βασικές συναρτήσεις που μπορείτε να **καλέσετε** μέσα στον processor

▶ void send(MessageHandle&)

Στέλνει ένα μήνυμα στους γείτονες π.χ. :

```
send( new MyMessage() );
```

▶ void set_state(MessageHandle&)

Θέτει την κατάσταση του processor π.χ. :

```
set_state(Active)
set_state(Sleep)
set_state(Inactive)
```

▶ int simulation_round()

Επιστρέφει τον γύρο στον οποίο βρίσκεται η εξομοίωση

▶ int owner_w().id()

Επιστρέφει το id του κόμβου

▶ int owner_w().label()

Επιστρέφει το label του κόμβου



Τα μηνύματα στο Shawn

- ▶ Όλα τα μηνύματα που στέλνουμε είναι κλάσεις και βρίσκονται συνήθως στα αρχεία <name>_message.{h/cpp}. Και κληρονομούν από την κλάση shawn Message

```
class MyMessage
    : public shawn::Message
{
public:
    MyMessage();
    virtual ~MyMessage();
};
```

- ▶ Κάθε πρωτόκολλο υλοποιεί τα δικά του μηνύματα, δηλαδή διαφορετικές κλάσεις
- ▶ Κάθε processor κάνει cast το γενικό μήνυμα που λαμβάνει στην process_message(MessageHandle&) για να πάρει το δικό του μήνυμα

```
const MyMessage* mymsg =
dynamic_cast<const MyMessage*>( mh.get() );
```



Τα μηνύματα στο Shawn

- ▶ Το παρακάτω μήνυμα έχει μια μεταβλητή `id_` και μια συνάρτηση `id()` για να την διαβάσουμε
- ▶ Αποστολή ενός μηνύματος : `send(new FloodMaxMessage(3))`

Παράδειγμα κώδικα ενός μηνύματος (`floodmax_message.h`)

```
class FloodMaxMessage : public shawn::Message
{
public:
    FloodMaxMessage( int );
    virtual ~FloodMaxMessage ();

    inline int id( void ) const throw() {
        return id_;
    };
private:
    int id_;
};
```



Επισκόπηση

Ανασκόπηση

- Εκτέλεση μιας εξομοίωσης
- Πώς φτάνουμε μια νέα εφαρμογή στο shawn
- The Processor
- The Messages

JShawn

JShawn

Ασκήσεις

Ασκηση Μερøs B



JShawn (1 of 4)

- ▶ Μέχρι τώρα για να εκτελέσουμε τις εξομοιώσεις γράφουμε όλες τις παραμέτρους σε ένα `.conf` αρχείο
- ▶ Αλλά τι γίνεται σε περίπτωση που θέλουμε να έχουμε πιο πολύπλοκο έλεγχο των εξομοιώσεων;
- ▶ Για παράδειγμα έστω ότι χρειαζόμαστε να εκτελέσουμε 100 φορές μια εξομοίωση
 - ▶ for-loops
 - ▶ if-then
- ▶ Το JShawn είναι ένας διαφορετικός τρόπος για να γράφουμε config αρχεία που μας επιτρέπει πλήρη έλεγχο της εξομοίωσης



JShawn (2 of 4)

- ▶ Οι κύριες εντολές ενός JShawn config αρχείου είναι:
 1. `shawn.setGlobalVariable("variable", "value")` :
Θέτει μια μεταβλητή, είναι ισοδύναμο με `variable=value` στο κλασικό config αρχείο
 2. `shawn.runCommand("task-name", "extra parameters")` :
Εκτελεί ένα μια εντολή π.χ. ("`simulation`") με παραμέτρους
- ▶ Το JShawn υποστηρίζει πλήρη σύνταξη Java οπότε μπορείτε:
 - ▶ Να δηλώσετε μεταβλητές `nx int l=0`
 - ▶ Να τυπώσετε διαγνωσικά μηνύματα π.χ. `System.out.println("l = " + l)`
 - ▶ Να χρησιμοποιήσετε for-loops π.χ. `for(int l=0;l<10;l++)`



Παράδειγμα JShawn αρχείου

```
String processor_name = "helloworld";

shawn.runCommand('prepare_world', 'edge_model=simple
                                comm_model=disk_graph range=2' );

shawn.runCommand('rect_world', 'width=5 height=5
                                count=30 processors=' + processor_name);

shawn.runCommand('simulation', 'max_iterations=10');
```



- ▶ Για να ξεκινήσετε μια εξομίωση με το JShawn δίνεται την εντολή από τον φάκελο buildfiles:

Παράδειγμα εκτέλεσης JShawn αρχείου

```
java -jar ../jshawn-allinone.jar
-s ./shawn -b conf_file.jshawn
```



Επισκόπηση

Ανασκόπηση

Εκτέλεση μιας εξομίωσης
Πώς φτιάχνουμε μια νέα εφαρμογή στο shawn
The Processor
The Messages

JShawn

JShawn

Ασκήσεις

Ασκήση Μερος Β



Περιγραφή του προβλήματος

- ▶ Σε προηγούμενο εργαστήριο είχαμε υλοποιήσει ένα απλό καταναμημένο αλγόριθμο (FR02).
- ▶ Tree Routing Algorithm
 1. Ένας κόμβος (ο Gateway με label v0) κάνει flood στο δίκτυο ένα μήνυμα.
 2. Το flood μήνυμα περιέχει το label του κόμβου που το έστειλε και το hop-count από το Gateway
 3. Κάθε κόμβος που λαμβάνει το flood μήνυμα:
 - 3.1 αποθηκεύει το label του κόμβου από τον οποίο πήρε το μήνυμα (parent node) και το hop-count, μόνο αν το hop-count είναι μικρότερο από αυτό που ήδη έχει
 - 3.2 Μπαίνει σε κατάσταση connected
 4. Μετά στέλνουν ένα νέο flood μήνυμα που έχει αποστολέα τον ίδιο και hop-count= hop-count +1



Άσκηση Μερους Β

- ▶ Μπορείτε να κατεβάσετε την λύση του του πρώτου προβλήματος από εδώ
`ftp://carrot.cti.gr/fr02/fr02_partB.tar.bz2`
- ▶ Μεταφέρεται τα αρχεία `fr02-config.conf` και `fr02-topology.xml` στο φάκελο `shawn/buildfiles`
- ▶ Δημιουργήστε ένα νέο φάκελο `shawn/src/legacypapps/fr02`
- ▶ Μεταφέρεται υπόλοιπα αρχεία στον φάκελο `shawn/src/legacypapps/fr02`
- ▶ Από την κονσόλα μεταβείτε στον φάκελο `shawn/buildfiles`
- ▶ Δώστε την εντολή `csmake ../src` και μετά:
 - ▶ `c` , για να ενημερωθεί για τον νέο κώδικα που προσθέσατε
 - ▶ Ενεργοποιήστε την επιλογή `FR02` (πατώντας `enter`)
 - ▶ `c` , για να ενημερωθεί για τις αλλαγές
 - ▶ `g` , για να αποθηκεύσει τις αλλαγές
- ▶ Τέλος δώστε την εντολή `make` για να γίνει `compile` το `shawn`



Άσκηση Μερους Β

- ▶ Στο μέρος Β θα επεκτείνουμε την προηγούμενη άσκηση ώστε:
 - ▶ Κάθε κόμβος που μπαίνει στο δέντρο θα στέλνει σε κάθε γύρο ένα μήνυμα προς το Gateway
 - ▶ Το μήνυμα θα περιέχει το hop-count του κόμβου
 - ▶ Κάθε κόμβος θα στέλνει το μήνυμα με προορισμό τον πατέρα του στο δέντρο, μέχρι να φτάσει στην ρίζα
 - ▶ Τα μηνύματα είναι τύπου `Fr02HelloMessage`
 - ▶ Όταν η ρίζα λάβει ένα μήνυμα `Fr02HelloMessage` θα τυπώνει τον αποστολέα και το hop-count του
 - ▶ Προαιρετικά μπορείτε να υπολογίζετε και να τυπώνεται:
 1. Τον συνολικό αριθμό κόμβων που γνωρίζει η ρίζα
 2. Το μέγιστο, ελάχιστο και μέσο όρο του hop count των κόμβων



Άσκηση Μερους Β

- ▶ Για αυτή την άσκηση στα αρχεία `fr02_processor.cpp`, `fr02_processor.h` αφαιρέστε από σχόλια τις συναρτήσεις που αρχίζουν με :

```
// Uncomment on part 2 of the exercise
```
- ▶ Και υλοποιήστε την επιπλέον λογική του αλγορίθμου υλοποιώντας τις ήδη υπάρχουσες συναρτήσεις στα αρχεία `fr02_processor.cpp`, `fr02_processor.h`

