Distributed Computing and Systems
Chalmers university of technology

LYDIAN

Phuong Hoai Ha

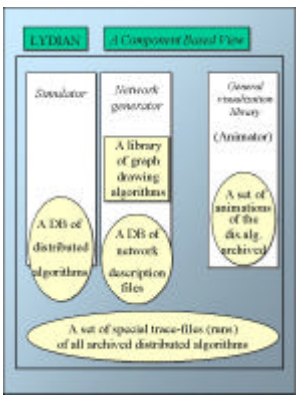
Introduction to Lab. assignments
April 6th, 2005

Schedule

- LYDIAN: Animation Environment for Learning Distributed Algorithms
  - Overview
  - Making new protocols
  - Making new networks
- POLKA: Animation Package
  - Animator class
  - View class
  - Objects

LYDIAN

- Distributed algorithms are complicated
  - Involve a large amount of data describing local state information and complex interactions between elements
- Pseudo-code description given in a book or technical paper is often not sufficient for the understanding of an algorithm
- Lydian is an educational tool for teaching distributed algorithms by allowing users to create their own experiments:
  - Creating/selecting a protocol
  - Describing the network
  - Selecting/adding animations



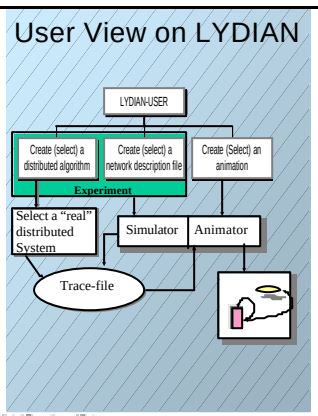
Components

All components are extensible

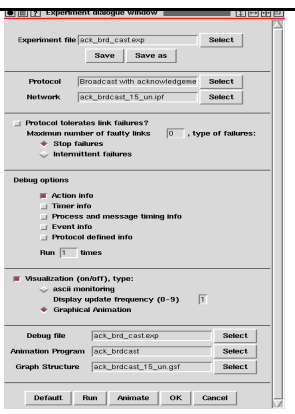
Accessed via a GUI

Can be replaced by new components

User View on LYDIAN



Demo



The Simulator DIAS

- Simulates any number of processes
- Processes communicate by sending messages
- The behaviour of processes can be defined by defining state transitions
  - State x Event -> function()
- States, messages and functions can be defined in pseudocode
- Simulator supports user defined timing behaviour for message transmission and local computation
- Supports simulation of communication failures for fault tolerance tests

Animator

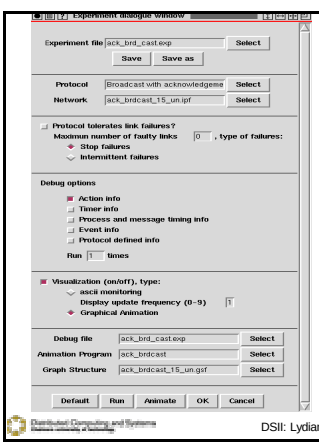
- For many of the provided protocols LYDIAN provides an animation
- The animation uses the output of the simulator (the tracefile) or alternatively online events of a real distributed system
- The main window always shows the commnunication graph as it was created by the user
- Different Animation aspects are supported by different animation windows which all evolve at the same time and selected by the user
- Further aspects concentrate on causalities, traffic, events and processes occupation in a computation

Getting started

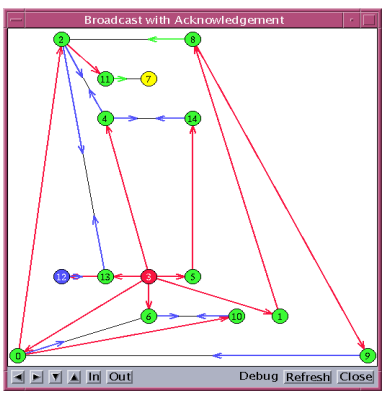
- Login to a Sun machine (remote1[,remote2, remote3, remote4].tekno.chalmers.se)
  - Right now Lydian runs only on Sun machines.
- *setenv LYDIANROOT /users/mdstud/phuong/LYDIAN/*
- *$LYDIANROOT/GUI/userinst ./*
- Look at some useful demonstrations
  - Broadcast with acknowledgement algorithm (file ack_brd_cast.exp)
  - Ricart and Agrawala's Resource Allocation algorithm (file ResAloc_RA_1.exp)
  - … (algorithm descriptions are stored at $LYDIANROOT/Html)

Broadcast with ACK algorithm

- An initiator process sends a broadcast message to all its neighbours
- If receiving the message for the first time, each process sends the message to all its adjacent processes except for its *parent.*
- If receiving the message again, processes reply with ACK.
- Each process waits for ACKs from its receivers. When receiving all the ACKs, it sends a ACK to its parent and stops.
- Algorithm is finished when the initiator received all ACKs



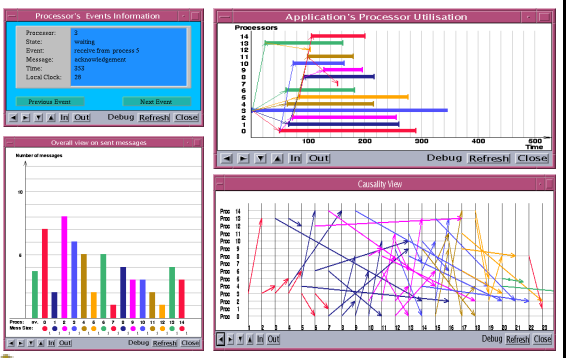
Experiment



Main view

- R initiator
- Y sleeping
- G waiting
- B done
- G BRD msg
- B ACK msg
- R spanning tree

Other animation aspects



Making new protocol

- Design

	$state_1$	$state_2$	…	$state_n$
$event_1$				
$event_2$				
…				
$event_m$				

An example: Broadcast with ACK

	SLEEPING	WAITING	DONE
BRD	•if(leaf) {reply with ACK; new_state==DONE;} •*parent* = sender; •send BRD to other adjacent processes; •store receivers in a set *ack*; •new_state = WAITING;	reply with ACK;	reply with ACK;
ACK	Never	•delete sender from *ack*; •if(*ack* == empty) and (*parent* != nil) {send ACK to parent; new_state = DONE} •if(*ack* == empty) and (*parent* == nil) **finished**;	Never

Protocol framework

- Implementation

```
typedef struct {
    /* all local variables must be declared here */
    …
    } REGISTERS;
REGISTERS *REG;

reg_alloc() {
    REG = (REGISTERS*)malloc(processes*sizeof(REGISTERS));
}

init() {
    /* all local variables must be initialized here */
    …
}

/* Below are your own procedures */
…
```


Useful references

- Message structure

typedef struct mess { …
int kind, value[10], from, port;} MESSAGES;

- Process Control Block (PCB)

typedef struct { …
int state, id, adjacents;} PCBS;

- Global variables

me: id of current process
CURMESS: current received message
new_state: the new state of current process
processes: the number of processes

- Message routines

MESSAGE* create_message()
send_to(MESSAGE *m, int port)

- List handling

init_list(LIST l)
insert_list(int e, LIST l)
delete_list(int e, LIST l)
int empty_list(LIST l)
int in_list(int e, LISt l)

- Timer manipulation

start_timer(TIMER1[2,3,4,5], int delay)
stop_timer(TIMER1[2,3,4,5])

- Miscellaneous

simul_end() /*stop simulation*/
init_event(int pid) /*send INITPROTOCOL event to pid*/
debug("f", "%d %s …", _value, _string, …)
int uniform(int lower, int upper) /*random*/

Ex: Broadcast with ACK (cont.)

```
typedef struct {
    int parent;
    LIST ack;
    } REGISTERS;
REGISTERS *REG;

reg_alloc() {
    REG = (REGISTERS*)malloc(processes*sizeof(REGISTERS));
}

init() {
    for(i = 0; i < processes; i++) {
        REG[i].parent = -1;
        init_list(REG[i].ack);
    }
}
```

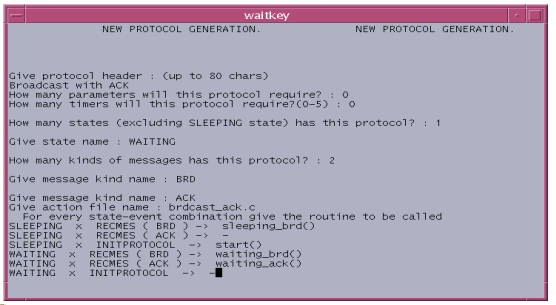
Ex: Broadcast with ACK (cont.)

```
sleeping_brd() {
    if( PCB[me].adjacents == 1) {
      mess =create_message();}
      mess->kind = ACK;
      send_to(mess, CURMESS->port);
      new_state= DONE;}
    else {
      REG[me].parent = CURMESS->port;
      for( i=0; i< PCB[me].adjacents; i++ )
        if( i != REG[me].parent ) {
          insert_list( i, REG[me].ack);
          mess = create_message();
          mess->kind = BRD;
          send_to( mess, i); }
      new_state= WAITING;
    }}

waiting_brd() { /*done_brd()*/
    mess = create_message();
    mess->kind = ACK;
    send_to( mess, CURMESS->port);
}
```

```
waiting_ack() {
    delete_list( CURMESS->port, REG[me].ack );
    if( empty_list( REG[me].ack)) {
      if( REG[me].parent == -1) { /* initiator */
        simul_end(); return;}
      else { /* send ACK to its parent */
        mess = create_message();
        mess->kind = ACK;
        send_to( mess, REG[me].parent);
        new_state= DONE;
      }}}

start() { /* initiators start simulation*/
    if( PCB[me].adjacents == 0) {
      simul_end(); return;}
    for( i=0; i< PCB[me].adjacents; i++) {
      insert_list( i, REG[me].ack);
      mess =create_message();
      mess->kind = BRD;
      send_to( mess, i); }
    new_state= WAITING;}
```


Adding new protocols to LYDIAN

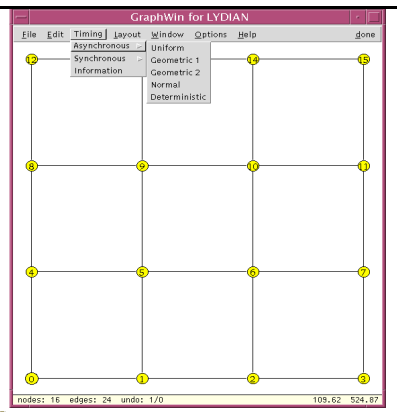
- Copy source file to directory your_installed_LYDIAN/DIAS



Making new networks on GraphWin

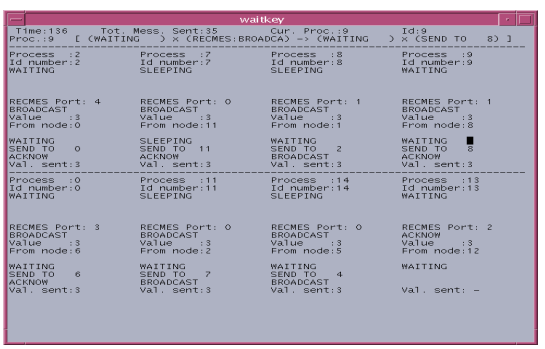
- Supports the creation of network description files
- Interconnection of processes is represented by a graph (nodes=processes; edges=links)
- Supports an easy creation of networks as well as an easy specification of timing parameters
- Supports a set of graph drawing algorithms for designing the layout of the network
- The created structure of the network can be used by an animation in order to create the same layout as it was defined by the user

- Notes:
  - GraphWin menu was changed. All functionalities for creating Lydian networks will appear by pressing button Network.
  - Use tool your_installed_LYDIAN/DIAS/nwconvert in order to convert file *.ipf generated by GraphWin to the correct format.



GraphWin demo

ASCII monitoring



More information

- http://www.cs.chalmers.se/~lydian
- LYDIAN Lab. in the Distributed Systems I course
  - http://www.cs.chalmers.se/~phuong/lydianLab.html

Schedule

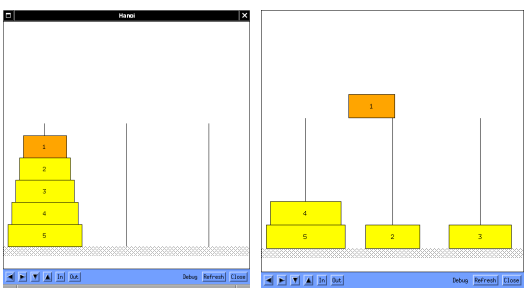
- LYDIAN: Animation Environment for Learning Distributed Algorithms
  - Overview
  - Making new protocols
  - Making new networks
- POLKA: Animation Package
  - Animator class
  - View class
  - Objects

POLKA

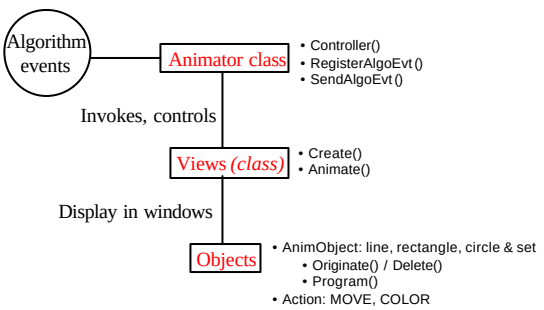
- A C++ library
- You can easy create your own ***animator*** class with following properties:
  - It is possible to ***send*** algorithm events to the class
  - The animator creates pictures according to the events
  - Many events can be shown at the same time
  - The animation can use many windows at the same time
  - The user can interact

Demo

- Hanoi tower



Key elements of POLKA



Animator class

- Each program/animation should have one Animator object
- Important functions:
  - Controller():
    - Check what event has just happened to call appropriate animation functions.
  - RegisterAlgoEvt(event name, parameters)
    - Register the event with its parameters.
  - SendAlgoEvt(event name, parameters)
    - Send an event with its parameters to the Animator.
    - Call Controller();

Example: Hanoi tower

```
class       MyAnimator : public Animator {
  public:
    int         Controller();

  private:
    HanoiState    state; /* View */
};

main()
{
  hanoi.RegisterAlgoEvt("Init", "d");
  hanoi.RegisterAlgoEvt("Move", "dd");
  hanoi.RegisterAlgoEvt("Done", NULL);

  cout << "How many disks?\n";
  int n;
  cin >> n;
  hanoi.SendAlgoEvt("Init", n);
  rec(n, 0, 2);
  hanoi.SendAlgoEvt("Done");
}
```

```
int MyAnimator :: Controller()
{
  int         retval = 0;

  if (!strcmp(AlgoEvtName, "Move")) {
    retval = state.Move(AnimInts[0], AnimInts[1]);
  } else if (!strcmp(AlgoEvtName, "Init")) {
    retval = state.Init(AnimInts[0]);
  } else if (!strcmp(AlgoEvtName, "Done")) {
    state.Done();
  }
  return (retval);
}

HanoiState :: Init( int n) { … }
HanoiState :: Move( int from, int to) { … }
HanoiState :: Done() { … }
```

View class

- A graphical perspective of a program.
  - Each view has its own animation time (frame number).
- Important functions:
  - Create()
    - create a X window containing the View.
  - Animate(start, length)
    - create a set of animation frame starting with *time*=*start* for *length* frames.
  - Refresh()
    - refresh the View's window.

AnimObjects class

- Graphical objects that are visible in an animation window.
  - Subclasses:
    - Line, Rectangle, Circle, Text, etc.
    - Set: a reference to a list of other AnimObjects.
  - Common fields:
    - an associated View, visibility, x and y locations.
- Important functions:
  - Originate(time)
    - add the AnimObject to the associated View since frame *time*
  - Delete(time)
  - Program(time, Action)
    - schedule *Action* to happen to the object at frame *time.*
    - return the duration (#frames) the *Action* will take place.
  - Other useful functions:
    - StoreData(), RetrieveData(): store data within each AnimObject, FILO.

Action

- A modification to an AnimObject
  - Position, size, color, etc.
  - Consist of
    1. An action type: MOVE, COLOR, RESIZE, etc.
    2. A path (sequence of steps)
- Important functions:
  - Interpolate(factor)
    - create an Action whose #offsets is modified by factor *factor.*
  - a1.Concatenate(Action *a2)
    - create an Action (*a1 Å a2)*

Framework

```
class MyView : public View {
    public:
        <animation functions>
    private:
        <animation objects>
}

class MyAnimator : public Animator {
    public:
        Controller();
    private
        MyView myview;
}

MyAnimator  myanimator;
```

```
controller() {
    <call functions corresponding to events>;
}

<define myview's animation functions>

main() {
    myanimator.RegisterAlgoEvt(<events>);
    <generate events>;
    myanimator.SendAlgoEvt(<events>);
}
```


Running an example

- Directory:

  /users/mdstud/phuong/DSII/demo/polka
  - Animation program for Hanoi tower:
    - *hanoi.C, hanoi.H, hanoiscenes.C*
  - Compile:
    - *Makefile*
  - Using the *Makefile* to compile your own animation program.
- POLKA documents:

  /users/mdstud/tsigas/DSII/polka/Doc/:
  - *polkadoc-lite.ps, polkadoc.ps*

References

- John T. Stasko. POLKA Animation Designer's Package, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280
- http:/www.cs.chalmers.se/~lydian