Distributing trust verification to increase application performance

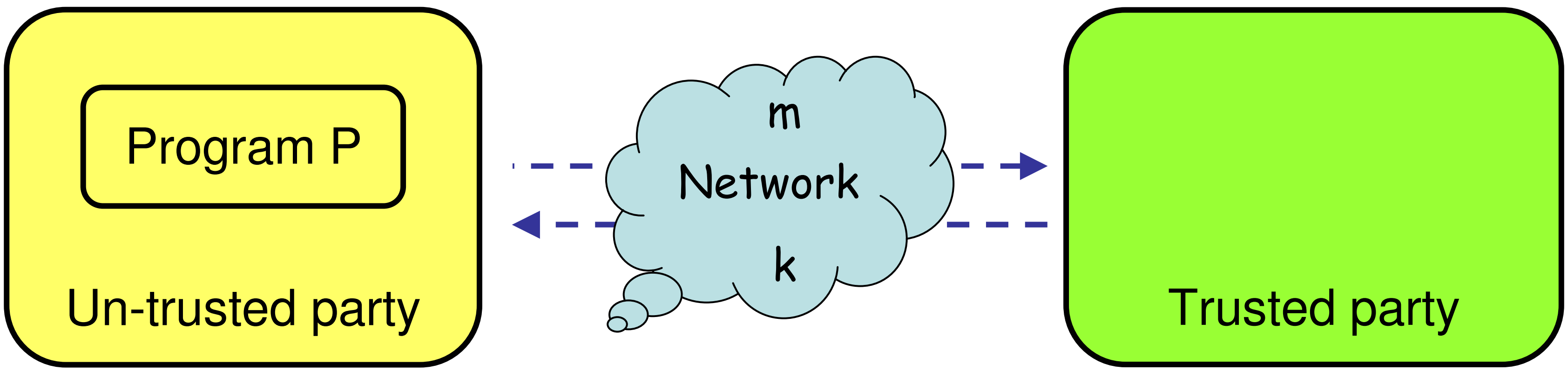
Ceccato Mariano[1],
Jasvir Nagra[2],
Paolo Tonella[1]

[1]Fondazione Bruno Kessler-IRST, Trento, Italy
[2]University of Trento, Italy

Problem definition

- Network application, that needs a services by the trusted party.
- Trusted party means to deliver the services only to clients that can be trustred.

- s: state of the program P
- m = f(s)
- k = g(m)
  = g(f(s))



Problem definition

P is a valid state:

A(s) = true

P is entrusted:

E(m) = true

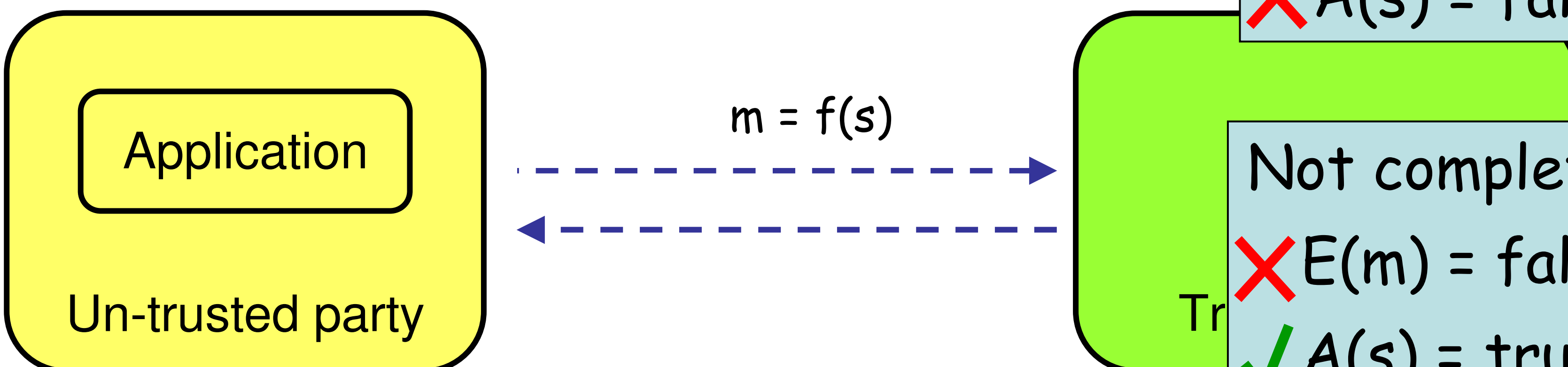
Remote entrusting problem:

E(m) ⇔ A(s)

Not sound:

✓ E(m) = true

✗ A(s) = false



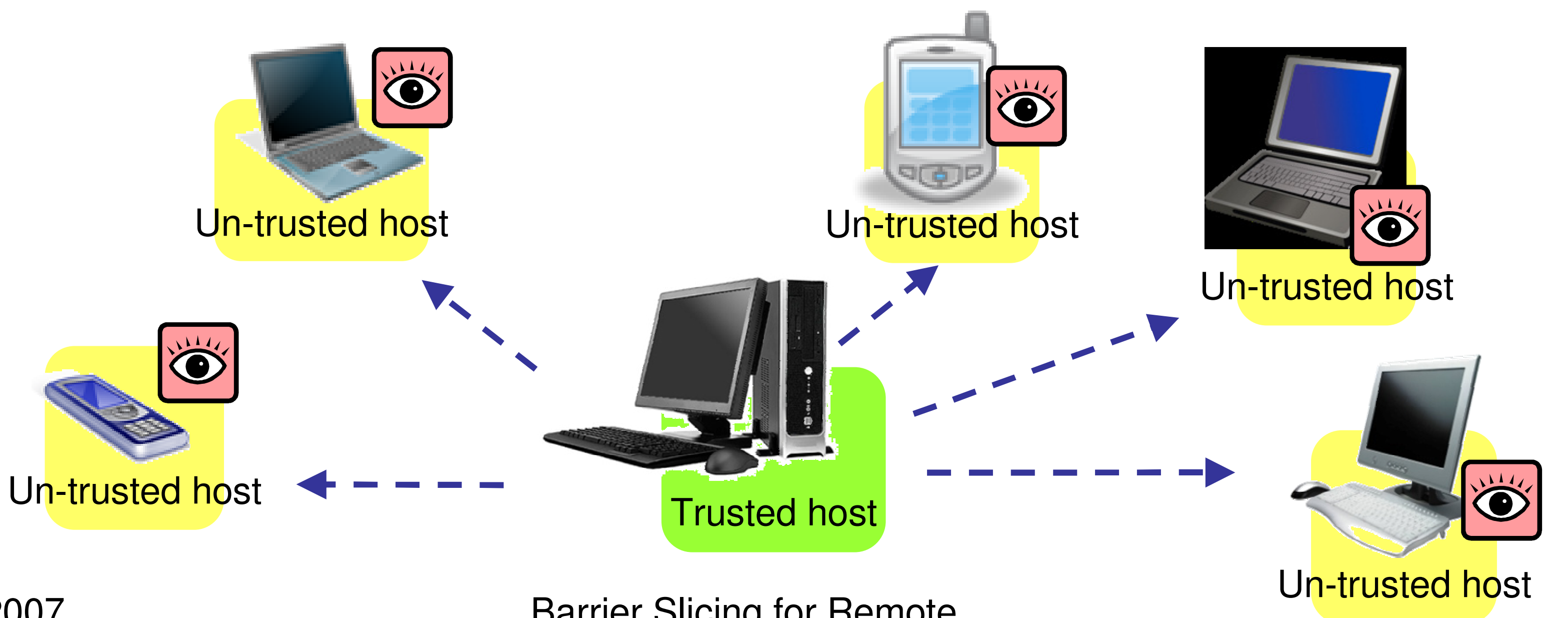
Not complete:

✗ E(m) = false

✓ A(s) = true

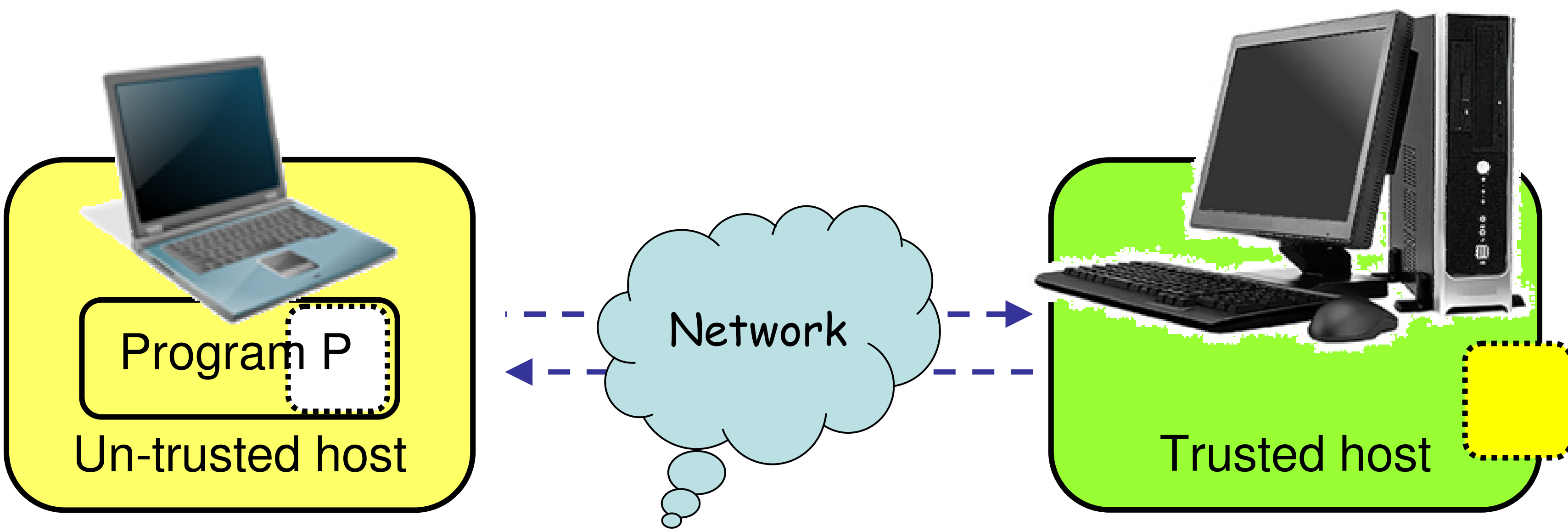
Remote software trusting

- *Remote software authentication*: ensuring a trusted machine (server) that an un-trusted host (client) is running a “healthy” version of a program;
- The server is willing to deliver a given services only to clients that prove to be “healthy”;
  - The program is unadulterated.
  - It is executed on top of unadulterated HW/SW.
  - The execution process is not manipulated externally.



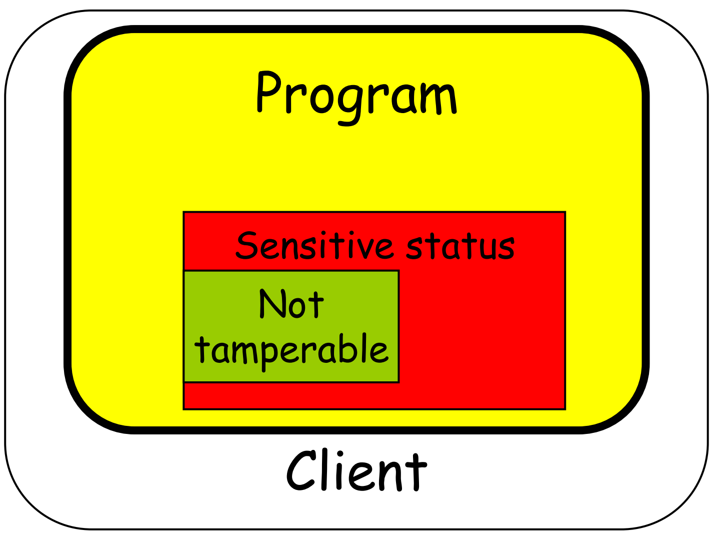
Previous slicing approach

- Remove a portion of the program to protect and run it on the server.
  - Trade off between security and performances



Program state partition

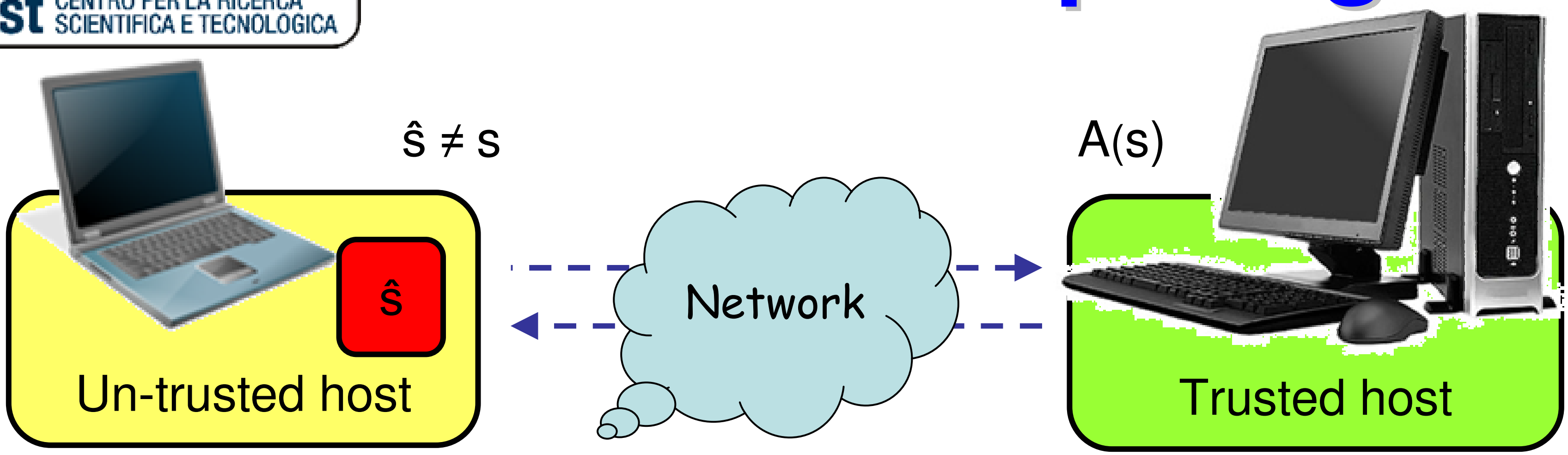
- There is a limited status (set of program variables) in an application that we are interested in protecting.
- A sub-portion of this state ($s_{|safe}$) can not modified by the user, otherwise
  - The client would receive a not-usable service or
  - The server would notice it



$$s = s_{|safe} \cup s_{|unsafe}$$

$$A(s) = A_{safe}(s_{|safe}) \wedge A_{unsafe}(s_{|unsafe})$$

State tampering



$\hat{s}_{|safe}$ is sent:

- $A_{safe}(\hat{s}_{|safe}) = false$,
- tampering is detected

$s_{|safe}$ ($!= \hat{s}_{|safe}$) is sent:

- $A_{safe}(s_{|safe}) = true$,
- Service is not usable
- Tampering is useless

$$\hat{s} = \hat{s}_{|safe} \cup \hat{s}_{|unsafe}$$

$$A(s) = A_{safe}(s_{|safe}) \wedge A_{unsafe}(s_{|unsafe})$$

Program slice

- Set of variables that we are interested in protecting.
- We remove those variable from the client.
- The (executable) slice is replicated into the server where it can be executed safely.

```
time2 = System.currentTimeMillis();
double delta = speed * (time2 - time);
x = x + delta * cos(direction);
y = y + delta * sin(direction);
Server.sendPosition(x,y);
if (track.isInBox(x, y)){
        gas = maxGas;
        lastFuel = time2;
}
else {
        gas = maxGas - (int) (time2-lastFuel);
        if (gas < 0){
                gas = 0;
                if (speed > maxSpeed /10)
                        speed = maxSpeed /10;
                else if (speed < minSpeed/10)
                        speed = minSpeed/10;
        }
}
time = time2;
```

Barrier slice

- Subset of variables that can not modified by the user, otherwise either:
  - the client would receive a not-usable service, or
  - the server would notice it (using assertions)
- They can be used as **barriers** and block the dependency propagation when slicing (Krinke, scam 2003)

```
time2 = System.currentTimeMillis();
double delta = speed * (time2 – time);
x = x + delta * cos(direction);
y = y + delta * sin(direction);
Server.sendPosition(x,y);
if (track.isInBox(x, y)){
        gas = maxGas;
        lastFuel = time2;
}
else {
        gas = maxGas - (int) (time2-lastFuel);
        if (gas < 0) {
                gas = 0;
                if (speed > maxSpeed /10)
                        speed = maxSpeed /10;
                else if (speed < minSpeed/10)
                        speed = minSpeed/10;
        }
}
time = time2;
```

safe

un-safe

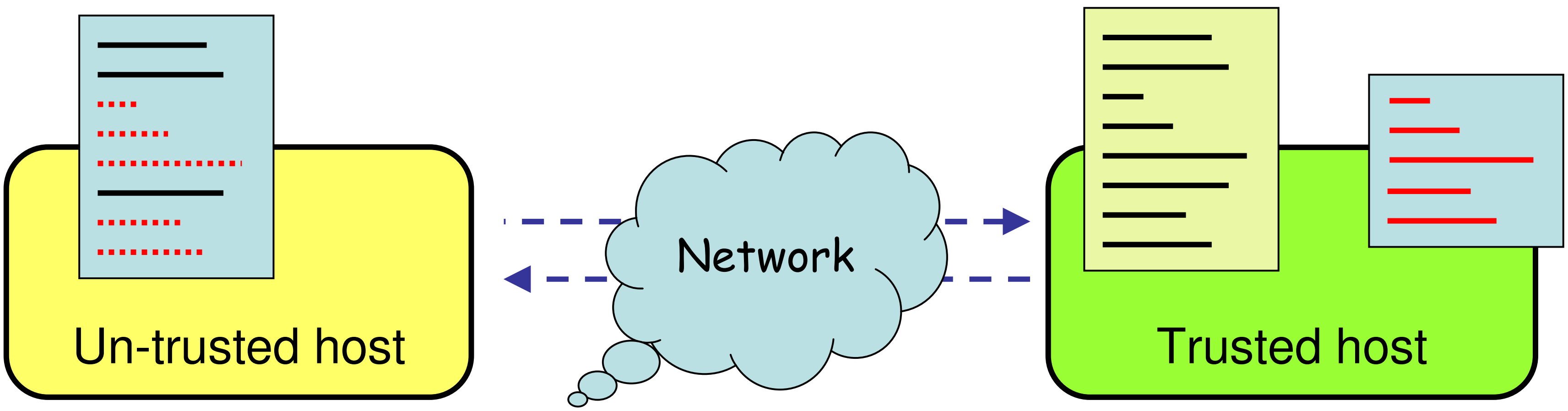
Program transformation

Un-trusted host:

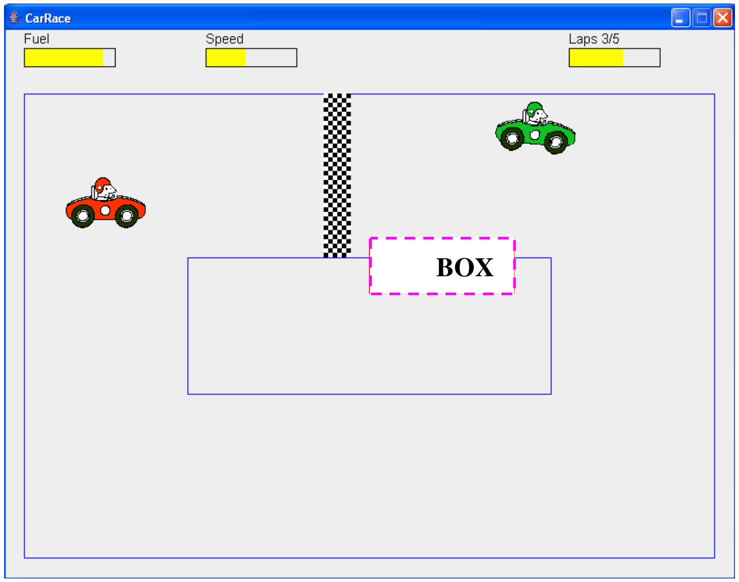
- X ∈ un-safe
- X **uses** are removed from the program;
- They are replaced by a query to get the actual value over the network;
- X **defs** are replaced by synchronization statements.
- Some optimizations…

Trusted host:

- A barrier-slice is run for each served host;
- Client validity is continuously verified (assertions);
- X values are provided as required;
- Synchronization with the un-trusted hosts.



Example: CarRace



Position

Number of Laps

Fuel

Speed

Original client	Slice	Barrier slice
858	185	120 (-65)
	22%	14% (-35%)

Performances

Non-optimized:

- Very small delay between command and car response.

Optimized:

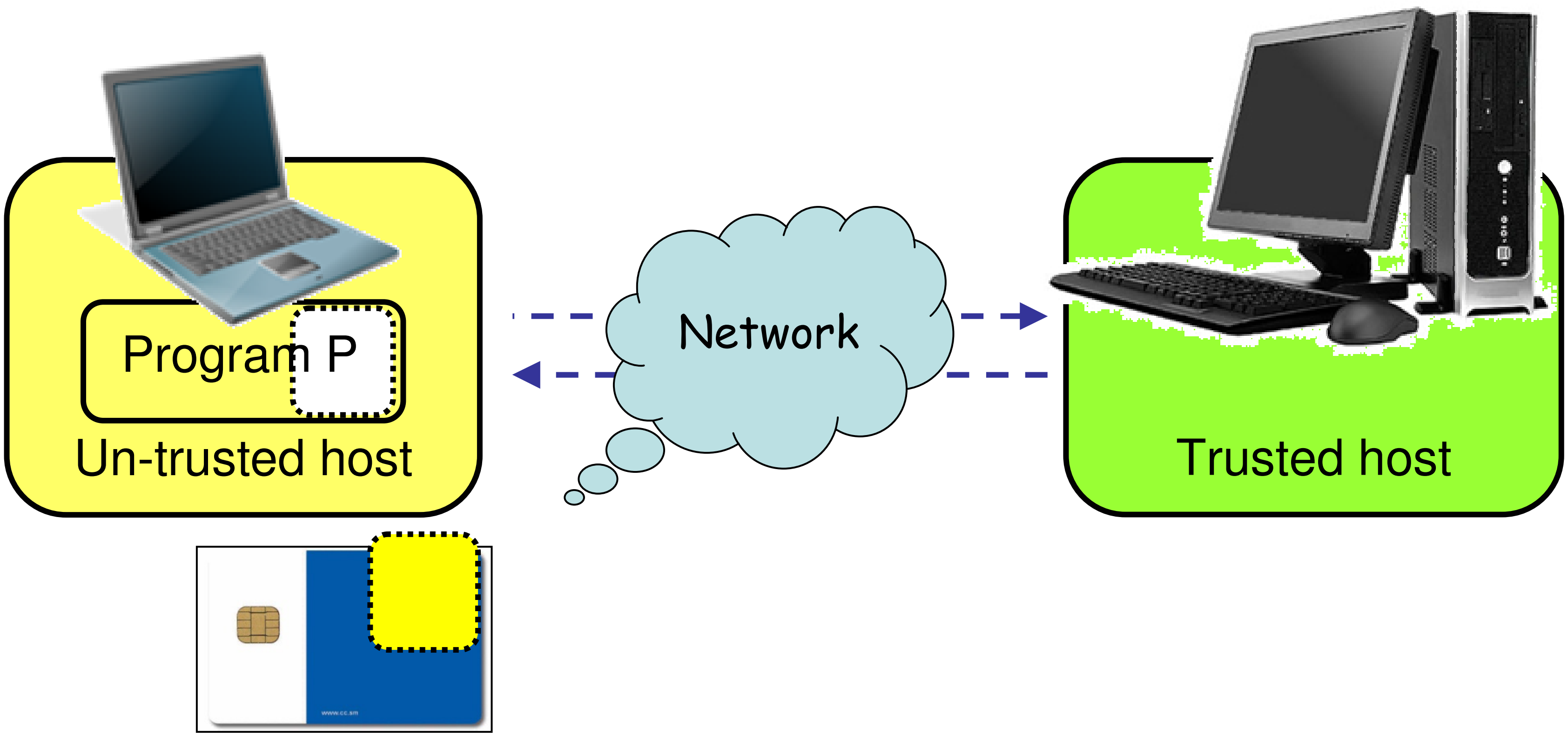
- No noticeable performance difference observed by the player.

Communication overhead:

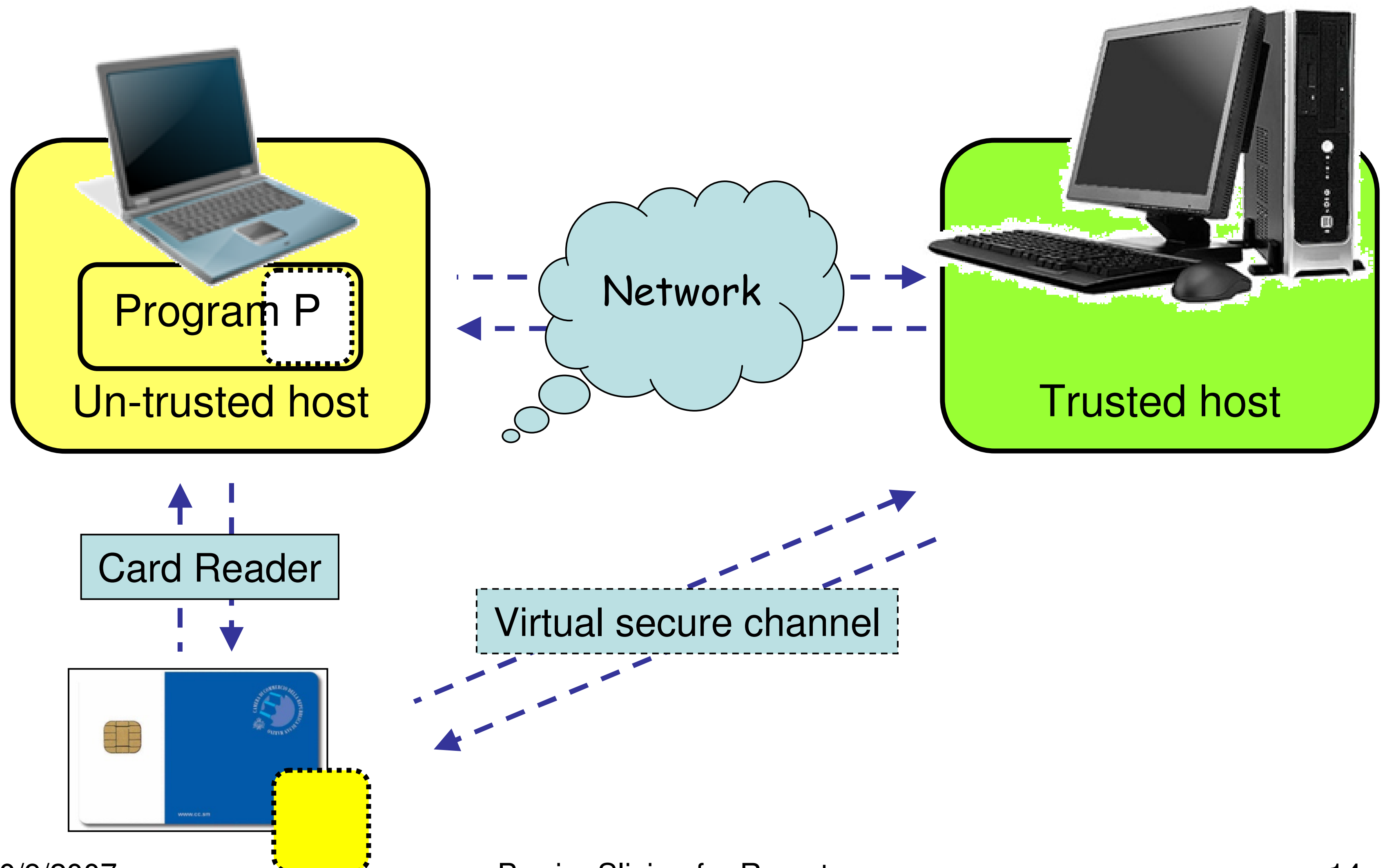
- Messages increase due to synchronization and delivery of $x \in$ un-safe

	Regular messages	Trust messaged	Increase
Sent	1174	5910	5.03
Received	1172	5910	5.04

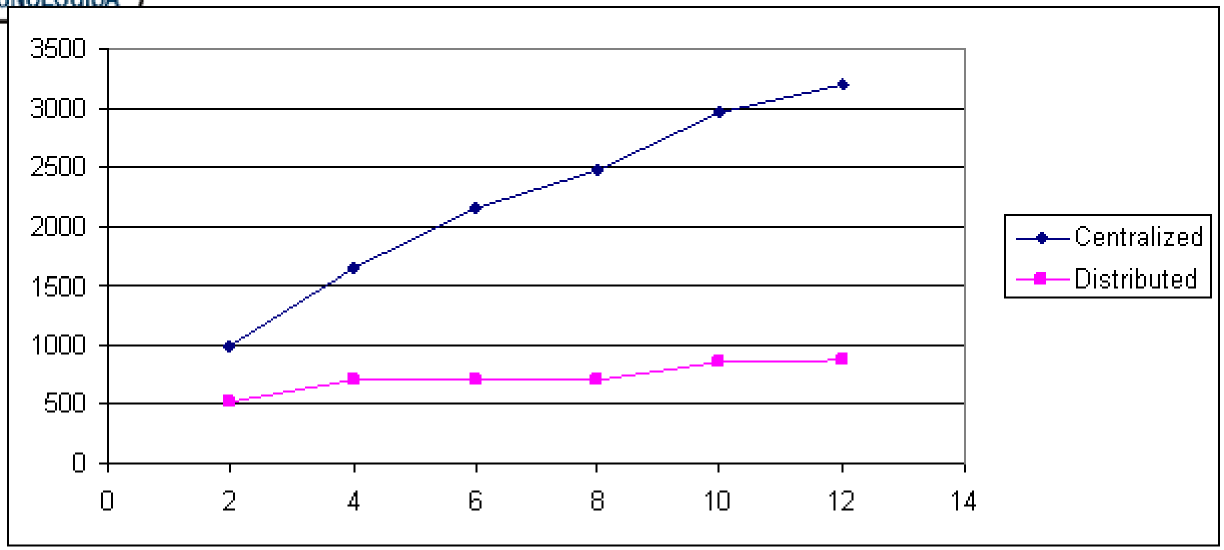
Distributed architecture



Distributed architecture

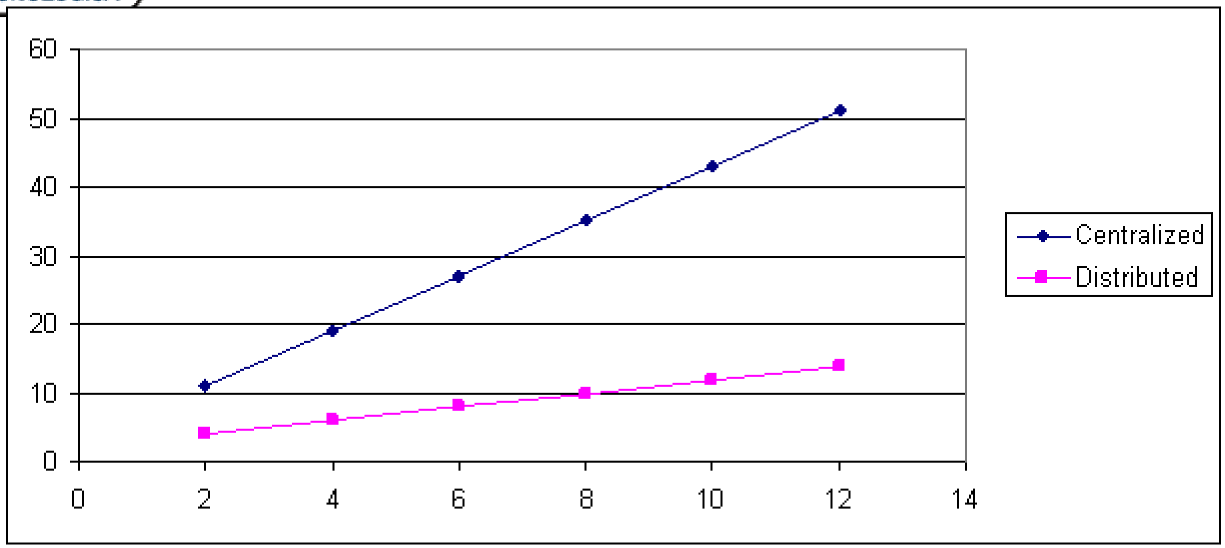


Memory scalability



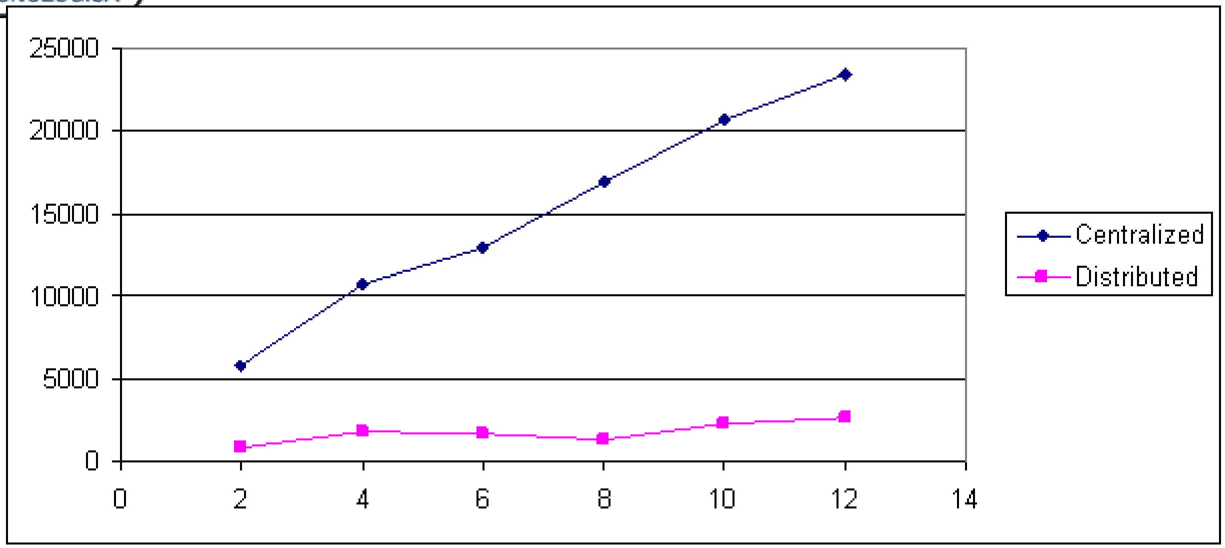
- For two clients the memory requirements is the double.
- 220 Vs 32 bytes per each connected client (15%)
- 2325 Vs 820 bytes in the heap space
- Slice requires less than 25% of the application CPU time

Threads scalability



- 4 Vs 1 new thread per connected client (25%)

Network scalability



- Distributed architecture exchanges the same number of message as the original (not protected) application.
- 145 Vs 1743 exchanged messages for each new connected client (8%).

Open challenges

- How to run multi-thread applications on a smart-card;
- Limited memory and runtime capabilities of smart cards;
- Architectural differences between JVM and SM-JVM (security manager, primitive types, libraries, etc.).

Ongoing works

- Automatic support for the identification of the secure and un-secure variables;
- Apply the barrier slicing to bigger test cases to perform overhead measurements;
- Integrate our approach with code obfuscation to shrink the portion of code to move on the card.