Adaptive Data Integrity through Dynamically Redundant Data Structures

Vincenzo De Florio  Chris Blondia

University of Antwerp
Performance Analysis of Telecommunication Systems group
http://www.pats.ua.ac.be
Interdisciplinary institute for BroadBand Technology
http://www.ibbt.be

Third International Workshop Dependability Aspects on Data Warehousing and Mining Applications (DAWAM 2008)
Outline

The Problem

Our Solution

Our Prototype

Preliminary results

Conclusions
A well-known result by Shannon tells us that, from any unreliable channel, it is possible to set up a more reliable channel by increasing the degree of information redundancy.

This is the same in fault-tolerance: A service that needs to be reliable in a disturbed environment requires a certain degree of redundancy (in time, information, design, or hardware) to cope with the disturbances.

OK, but how much redundancy?
A well-known result by Shannon tells us that, from any unreliable channel, it is possible to set up a more reliable channel by increasing the degree of information redundancy. This is the same in fault-tolerance: A service that needs to be reliable in a disturbed environment requires a certain degree of redundancy (in time, information, design, or hardware) to cope with the disturbances.

OK, but how much redundancy?
A well-known result by Shannon tells us that, from any unreliable channel, it is possible to set up a more reliable channel by increasing the degree of information redundancy. This is the same in fault-tolerance: A service that needs to be reliable in a disturbed environment requires a certain degree of redundancy (in time, information, design, or hardware) to cope with the disturbances. OK, but how much redundancy?
The common strategy: Fixed choice, dependent on the context. This translates into two risks, namely

1. overshooting: over-dimensioning the design with respect to the actual threat being experienced
2. undershooting: underestimating the threat in view of an economy of resources.

The chance of being precisely on the border is slim

(Another problem is what one does with this redundancy of course—weak vs strong failure semantics etc...)
The common strategy: Fixed choice, dependent on the context.
This translates into two risks, namely

1. **overshooting**: over-dimensioning the design with respect to the actual threat being experienced
2. **undershooting**: underestimating the threat in view of an economy of resources.

The chance of being precisely on the border is slim

(Another problem is what one does with this redundancy of course—weak vs strong failure semantics etc...)
The problem

The common strategy: Fixed choice, dependent on the context. This translates into two risks, namely

1. overshooting: over-dimensioning the design with respect to the actual threat being experienced
2. undershooting: underestimating the threat in view of an economy of resources.

The chance of being precisely on the border is slim.

(Another problem is what one does with this redundancy of course—weak vs strong failure semantics etc...)
The common strategy: Fixed choice, dependent on the context. This translates into two risks, namely

1. **overshooting**: over-dimensioning the design with respect to the actual threat being experienced
2. **undershooting**: underestimating the threat in view of an economy of resources.

The chance of being precisely on the border is slim

(Another problem is what one does with this redundancy of course—weak vs strong failure semantics etc...)
The Problem

The common strategy: Fixed choice, dependent on the context. This translates into two risks, namely

1. overshooting: over-dimensioning the design with respect to the actual threat being experienced

2. undershooting: underestimating the threat in view of an economy of resources.

The chance of being precisely on the border is slim

(Another problem is what one does with this redundancy of course—weak vs strong failure semantics etc...)
Wrong choices at this point can lead to
- either unpractical, too costly designs
- or cheap but vulnerable provisions.

In more formal terms, the problem is the static fault model assumption and the lack of adaptivity.
Let us focus on **integrity of data structures** and consider **redundant data structures**.

Redundancy and voting are used to protect memory from possible transient or permanent corruptions.

Again, the common strategy is a static fault model assumption. E.g. “during any mission, up to 1 fault shall affect the replicas” ⇒ using a 3-redundant cell to address the worst possible scenario. Again, a case of the undershooting / overshooting dilemma.
Let us focus on integrity of data structures and consider redundant data structures.
Redundancy and voting are used to protect memory from possible transient or permanent corruptions.
Again, the common strategy is a static fault model assumption. E.g. “during any mission, up to 1 fault shall affect the replicas” ⇒ using a 3-redundant cell to address the worst possible scenario. Again, a case of the undershooting / overshooting dilemma.
An example

Let us focus on integrity of data structures and consider redundant data structures.
Redundancy and voting are used to protect memory from possible transient or permanent corruptions.
Again, the common strategy is a static fault model assumption.
E.g. “during any mission, up to 1 fault shall affect the replicas” ⇒ using a 3-redundant cell to address the worst possible scenario.
Again, a case of the undershooting / overshooting dilemma.
An example

Let us focus on **integrity of data structures** and consider **redundant data structures**.

Redundancy and voting are used to protect memory from possible transient or permanent corruptions.

Again, the common strategy is a static fault model assumption. E.g. “during any mission, up to 1 fault shall affect the replicas” ⇒ using a 3-redundant cell to address the worst possible scenario.

Again, a case of the undershooting / overshooting dilemma.
Let us focus on **integrity of data structures** and consider redundant data structures. Redundancy and voting are used to protect memory from possible transient or permanent corruptions. Again, the common strategy is a static fault model assumption. E.g. “during any mission, up to 1 fault shall affect the replicas” ⇒ using a 3-redundant cell to address the worst possible scenario. Again, a case of the undershooting / overshooting dilemma.
Is this the right approach?

Does it make sense in the first place to fix, once and for all, a set of possible conditions affecting e.g. memory modules?
Our starting point

Is this the right approach?
Does it make sense in the first place to fix, once and for all, a set of possible conditions affecting e.g. memory modules?
Our starting point

We think not
E.g., memory modules: is it CMOS- or SDRAM-based? Big
difference here

— From *bit flips* to *single-event effects*.

Is the environment stable?

- Mobile services $\Rightarrow$ changing environments;
- External events may affect e.g. temperature, radiation,
electro-magnetic interference, cosmic rays, . . .
Our starting point

We think not
E.g., memory modules: is it CMOS- or SDRAM-based? Big difference here

— From *bit flips* to *single-event effects*.

Is the environment stable?

- Mobile services ⇒ changing environments;
- External events may affect e.g. temperature, radiation, electro-magnetic interference, cosmic rays, …
Our starting point

We think not
E.g., memory modules: is it CMOS- or SDRAM-based? Big difference here

— From bit flips to single-event effects.

Is the environment stable?

- Mobile services ⇒ changing environments;
- External events may affect e.g. temperature, radiation, electro-magnetic interference, cosmic rays, ...
Our solution

We should *consider the nature of faults as a dynamic system*, i.e., a system evolving in time, and express the fault model as a function $F(t)$.

Consequently, we believe that any fault-tolerance provision that be able to solve the over-dimensioning / under-dimensioning dilemma should make use of an *adaptative feedback loop*. In such loop redundancy would be allocated according to the measured values of $F(t)$, obtained by monitoring a set of meaningful environmental variables.
Our solution

We should *consider the nature of faults as a dynamic system*, i.e., a system evolving in time, and express the fault model as a function $F(t)$.

Consequently, we believe that any fault-tolerance provision that be able to solve the over-dimensioning / under-dimensioning dilemma should make use of an *adaptative feedback loop*. In such loop redundancy would be allocated according to the measured values of $F(t)$, obtained by monitoring a set of meaningful environmental variables.
Our solution

We should consider the nature of faults as a dynamic system, i.e., a system evolving in time, and express the fault model as a function $F(t)$. Consequently, we believe that any fault-tolerance provision that be able to solve the over-dimensioning / under-dimensioning dilemma should make use of an adaptative feedback loop.

In such loop redundancy would be allocated according to the measured values of $F(t)$, obtained by monitoring a set of meaningful environmental variables.
We should *consider the nature of faults as a dynamic system*, i.e., a system evolving in time, and express the fault model as a function $F(t)$. Consequently, we believe that any fault-tolerance provision that be able to solve the over-dimensioning / under-dimensioning dilemma should make use of an *adaptative feedback loop*. In such loop redundancy would be allocated according to the measured values of $F(t)$, obtained by monitoring a set of meaningful environmental variables.
Our solution

An example: Dynamically redundant data structures
A tool that allows designers to make use of adaptively redundant data structures with commodity programming languages.

Redundant data structures in which the degree of redundancy is not fixed once and for all at design time, but rather it changes dynamically with respect to the disturbances experienced during the run time.
An example: Dynamically redundant data structures
A tool that allows designers to make use of adaptively redundant data structures with commodity programming languages. Redundant data structures in which the degree of redundancy is not fixed once and for all at design time, but rather it changes dynamically with respect to the disturbances experienced during the run time.
Redundant variables

- A variable can be tagged as “redundant”
- Redundant variables are variables whose contents get replicated several times so as to protect them from memory faults
- Writing to a redundant variable means writing to a number of replicas, located “somewhere” and according to some strategy
- Reading from a redundant variable actually translates in reading from each of its cells and performing majority voting
- The result of this process is monitored by a special process, which we call Redundance. Redundance measures the amount of votes that differ from the majority vote, and uses this as a measure of the disturbance in the surrounding environment
Redundant variables

- A variable can be tagged as “redundant”
- Redundant variables are variables whose contents get replicated several times so as to protect them from memory faults
- Writing to a redundant variable means writing to a number of replicas, located “somewhere” and according to some strategy
- Reading from a redundant variable actually translates in reading from each of its cells and performing majority voting
- The result of this process is monitored by a special process, which we call Redundance. Redundance measures the amount of votes that differ from the majority vote, and uses this as a measure of the disturbance in the surrounding environment
Redundant variables

- A variable can be tagged as “*redundant*”
- Redundant variables are variables whose contents get replicated several times so as to protect them from memory faults
- Writing to a redundant variable means writing to a number of replicas, located “somewhere” and according to some strategy
- Reading from a redundant variable actually translates in reading from each of its cells and performing majority voting
- The result of this process is monitored by a special process, which we call Redundance. Redundance measures the amount of votes that differ from the majority vote, and uses this as a measure of the disturbance in the surrounding environment
Redundant variables

- A variable can be tagged as "redundant"
- Redundant variables are variables whose contents get replicated several times so as to protect them from memory faults
- Writing to a redundant variable means writing to a number of replicas, located “somewhere” and according to some strategy
- Reading from a redundant variable actually translates in reading from each of its cells and performing majority voting
- The result of this process is monitored by a special process, which we call Redundance. Redundance measures the amount of votes that differ from the majority vote, and uses this as a measure of the disturbance in the surrounding environment
**Redundant variables**

- Under normal situation, Redundance triplicates the memory cells of redundant variables. This corresponds to tolerating up to one memory fault in the cells associated to a redundant variable.

- Under more critical situations, the amount of redundancy must change.

- This is indeed what takes place:
  - The component that manages redundant variables declares the variable “ref_t int redundancy”.
  - The latter is set asynchronously by the Redundance process, which adjusts it so as to represent the “ideal” degree of redundancy with respect to the current disturbances.

- Implementation issues are shown in the paper.
Redundant variables

- Under normal situation, Redundance triplicates the memory cells of redundant variables. This corresponds to tolerating up to one memory fault in the cells associated to a redundant variable.
- Under more critical situations, the amount of redundancy must change.
- This is indeed what takes place:
  - The component that manages redundant variables declares the variable “ref_t int redundance”.
  - The latter is set asynchronously by the Redundance process, which adjusts it so as to represent the “ideal” degree of redundancy with respect to the current disturbances.
- Implementation issues are shown in the paper.
Redundant variables

- Under normal situation, Redundance triplicates the memory cells of redundant variables. This corresponds to tolerating up to one memory fault in the cells associated to a redundant variable.
- Under more critical situations, the amount of redundancy must change.
- This is indeed what takes place:
  - The component that manages redundant variables declares the variable “ref_t int redundancy”.
  - The latter is set asynchronously by the Redundance process, which adjusts it so as to represent the “ideal” degree of redundancy with respect to the current disturbances.
- Implementation issues are shown in the paper.
Preliminary results

We built a scrambling tool to analyze the performance of our system.
We simulated bit flips and bursts
We measured effectiveness and amount of redundancy actually used.
Preliminary results

All our experiments have been carried out with an array of 20000 redundant 4-byte cells and an allocation stride of 20 (that is, replicas of a same logical cell are spaced by 20 physical cells).

In all the reported experiments we run the following script:

```
SLEEP 1
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
SLEEP 3
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
END
```
Preliminary results

All our experiments have been carried out with an array of 20000 redundant 4-byte cells and an allocation stride of 20 (that is, replicas of a same logical cell are spaced by 20 physical cells).

In all the reported experiments we run the following script:

```
SLEEP 1
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
SLEEP 3
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
END
```
Concurrently with the execution of the above script, 1.5 million read accesses were performed in round robin across the array. The experiments record the number of scrambled cells and the number of read failures.
Concurrently with the execution of the above script, 1.5 million read accesses were performed in round robin across the array. The experiments record the number of scrambled cells and the number of read failures.
Experiment No. 1: Fixed, low redundancy.

```
$ scrambler faults.in 3 scrub noadaptive
Scrambler::sleep(1)
run 1
run 50001
run 100001
run 150001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 200001
run 250001
... lines omitted ...
run 650001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 700001
run 750001
... lines omitted ...
run 1500001
36734 scrambled cells, 193 failures, redundancy == 3
redundance 3: 1500001 runs
redundance 5: 0 runs
redundance 7: 0 runs
redundance 9: 0 runs
redundance 11: 0 runs
```
Experiment No. 2: Fixed, higher redundancy.

```plaintext
$ scrambler faults.in 5 scrub noadaptive
Scrambler::sleep(1)
run 1
run 50001
run 100001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 150001
... lines omitted ...
run 500001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 550001
... lines omitted ...
run 1500001
36734 scrambled cells, 0 failures, redundancy == 5
redundance 3: 0 runs
redundance 5: 1500001 runs
redundance 7: 0 runs
redundance 9: 0 runs
redundance 11: 0 runs
```
Experiment No. 3: Adaptive redundancy

$ scrambler faults.in 5 scrub adaptive
run 1
Scrambler::sleep(1)
run 50001
run 100001
run 150001
run 200001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 250001
... lines omitted...
run 600001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 650001
... lines omitted...
run 1500001
36734 scrambled cells, 1 failure, redundancy == 3
redundance 3: 1455404 runs
redundance 5: 6054 runs
redundance 7: 28967 runs
redundance 9: 9576 runs
redundance 11: 0 runs
Experiment No. 3: Adaptive redundancy

The first 100000 cycles of Experiment 3. Redundancy drops to its minimum after the first 1000 cycles.
Experiment No. 3: Adaptive redundancy: Zooming

A section of experiment 3 in which fault injection takes place.
We are in a changing world, so adaptive fault-tolerance provisions are a must.

We need to avoid the Horning syndrome:

"What is the most often overlooked risk in software engineering? That the environment will do something the designer never anticipated."

We need to include explicitly this aspect into the design, testing, and assessment processes, or else our services will not be able to cope with the threats of current and future environments.
Thank you for the attention

Questions?

vincenzo.deflorio@ua.ac.be
Thank you for the attention

Questions?
vincenzo.deflorio@ua.ac.be
Implementation Issues

- Core of the redundant variables architecture: A parser that translates the input source code into two source files:
  - An augmented version of the original code
  - A server-side, to monitor and drive the external devices (i.e., sensors and actuators)
Implementation Issues

```c
#include <stdio.h>
int main(void)
{
    int a;
    Ref_t int Redundance;
    redundant int myProtectedInteger;

    while (1) {
        sleep(5);
        printf("&Redundance == \%x, Redundance == \%d\n",
               &Redundance, Redundance);
        if (cpu > 90) break;
    }

tcpTxRate = 70;
myProtectedInteger = 1; // writes are replicated Redundance times
a = myProtectedInteger; // reads are managed through voting
```
int main(void)
{
    int a;
    /* Ref_t */ int tcpTxRate;                // 1
    /* Ref_t */ int Redundance;
    /* redundant */ int myProtectedInteger;

    reflex = aopen(acmp), rtypes = aopen(acmp);  // 2
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);

    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);

    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);

    pthread_create(&t, NULL, Server, (void*) reflex);  // 4

    while (1) {
        sleep(5);
        printf(" &cpu = %x, cpu = %d\n", &cpu, cpu);
        if (cpu > 90) break;
    }

    tcpTxRate = 70;               // 5
    CallTcpTxRate(&tcpTxRate);

    myProtectedInteger = 1;       // 6
    RedundantAssign_int(&myProtectedInteger);

    a = RedundantRead_int(&myProtectedInteger);  // 7
}

int main(void)
{
    int a;
    /* Ref_t */ int tcpTxRate;
    /* Ref_t */ int Redundance;
    /* redundant */ int myProtectedInteger;

    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);
    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);

    pthread_create(&t, NULL, Server, (void*) reflex);

    while (1) {
        sleep(5);
        printf("&cpu = %x, cpu = %x\n", &cpu, cpu);
        if (cpu > 90) break;
    }
    tcpTxRate = 70;
    CallTCPtxRate(&tcpTxRate);
    myProtectedInteger = 1;
    RedundantAssign_int(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger);
}

// 1: remove keywords

int main(void)
{
    int acl;
    /* Ref_t */ int tcpTxRate;
    /* Ref_t */ int Redundance;
    /* redundant */ int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);  // 3
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);
    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);
    pthread_create(&t, NULL, Server, (void*) reflex);  // 4

    while (1) {
        sleep(5);
        printf("\%s, cpu == %s", cpu == %d\n", &cpu, cpu);
        if (cpu > 90) break;
    }

tcpTxRate = 70;    // 5
CalltcpTxRate(&tcpTxRate);
myProtectedInteger = 1;       // 6
RedundantAssign_int(&myProtectedInteger);
a = RedundantRead_int(&myProtectedInteger);    // 7
}
Implementation Issues

```c
int main(void) {
    int a;
    /* Ref_t */ int tcpTxRate;
    /* Ref_t */ int Redundance;
    /* redundant */ int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate"); (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate"); (void*)129);
    awrite(reflex, "Redundance"); (void*)&Redundance);
    awrite(rtypes, "Redundance"); (void*)129);
    awrite(reflex, "myProtectedInteger"); (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger"); (void*)65);
    pthread_create(&t, NULL, Server, (void*) reflex); // 4

    while (1) {
        sleep(5);
        printf("&cpu == %x, cpu == %d
", &cpu, cpu);
        if (cpu > 90) break;
    }

tcpTxRate = 70;
CalltcpTxRate(&tcpTxRate); // 5
myProtectedInteger = 1;
RedundantAssign_int(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger); // 7
}
```

// 1: remove keywords
// 2: Open associative arrays
// 3: Associate identifiers to useful info
int main(void)
{
    int ai;
    /* float */  int tcpTxRate;
    /* float */  int Redundance;
    /* redundant */  int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp)
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);
    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);
    pthread_create(&t, NULL, Server, (void*) reflex);

    while (1) {
        sleep(5);
        printf("%x, cpu == %d\n", &cpu, cpu);
        if (cpu > 90) break;
    }

tcpTxRate = 70;  // 5
    CalltcpTxRate(&tcpTxRate);

    myProtectedInteger = 1;  // 6
    RedundantAssign_int(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger);  // 7
}
Implementation Issues

```c
int main(void) {
    int a;
    /* Ref_t */ int tcpTxRate;
    /* Ref_t */ int Redundance;
    /* redundant */ int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);

    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);
    pthread_create(&t, NULL, Server, (void*) reflex);

    while (1) {
        sleep(5);
        printf("%x, cpu == %d\n", &cpu, cpu);
        if (cpu > 90) break;
    }

    tcpTxRate = 70;
    CalltcpTxRate(&tcpTxRate);

    myProtectedInteger = 1;
    RedundantAssign_int(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger);
}
```

1: remove keywords
2: Open associative arrays
3: Associate identifiers to useful info
4: Create server thread
5: Writing to x implies calling Callx()
6
7
Implementation Issues

```c
int main(void)
{
    int a;
    // Ref_t */ int tcpTxRate;
    // Ref_t */ int Redundance;
    // redundant */ int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);
    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);
    pthread_create(&t, NULL, Server, (void*) reflex);

    while (1) {
        sleep(5);
        printf("%x, cpu = %d\n", &cpu, cpu);
        if (cpu > 90) break;
    }
    tcpTxRate = 70;
    CalltcpTxRate(&tcpTxRate);
    myProtectedInteger = 1;
    RedundantAssign_int(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger); // 7
}
```

1: remove keywords
2: Open associative arrays
3: Associate identifiers to useful info
4: Create server thread
5: Writing to x implies calling Callx()
6: Redundant write
int main(void)
{
    int a;
    int tcpTxRate;
    int Redundance;
    int myProtectedInteger;
    reflex = aopen(acmp), rtypes = aopen(acmp);
    awrite(reflex, "tcpTxRate", (void*)&tcpTxRate);
    awrite(rtypes, "tcpTxRate", (void*)129);
    awrite(reflex, "Redundance", (void*)&Redundance);
    awrite(rtypes, "Redundance", (void*)129);
    awrite(reflex, "myProtectedInteger", (void*)&myProtectedInteger);
    awrite(rtypes, "myProtectedInteger", (void*)65);
    pthread_create(&t, NULL, Server, (void*)reflex);
    while (1) {
        sleep(5);
        printf("%c", cpu = &x, cpu = &x
        if (cpu > 90) break;
    }
    tcpTxRate = 70;
    CallTcpTxRate(&tcpTxRate);
    myProtectedInteger = 1;
    ReadProtectedInt(&myProtectedInteger);
    a = RedundantRead_int(&myProtectedInteger);
}
Implementation Issues

- Functions “Call\(v(\&v)\)” must be supplied by the user.
- Functions “RedundantAssign\(_t(\&w)\)” and “RedundantRead\(_t(\&w)\)” are automatically generated through a template-like approach.
- The former performs a redundant write, the latter a redundant read plus majority voting.
- The “Server” thread is the code responsible to monitor and interface the external devices. Its algorithm is quite simple:
Implementation Issues

- Functions “Callv(&v)” must be supplied by the user
- Functions “RedundantAssign_t(&w)” and “RedundantRead_t(&w)” are automatically generated through a template-like approach
- The former performs a redundant write, the latter a redundant read plus majority voting.
- The “Server” thread is the code responsible to monitor and interface the external devices. Its algorithm is quite simple:
```
int Server(void)
{
    char sensor[80];    /* the name of the sensor */
    void *p;            /* the address of a reflective var */
    int type;           /* its type */
    mytypes_t object;   /* the new value of that varable */

    while (1) {
        /* wait for sensor update */
        if ( getValue(sensor, &object) == -1 ) continue;  // 1

        /* message received */
        p = (void*)aread(reflex, sensor); // 2
        type = (int) aread(rtypes, sensor);
        if (p != NULL) // 3
        {
            if (type & IsChar) *((char*)p) = object.c;
            if (type & IsShort) *((short*)p) = object.s;
            if (type & IsInt) *((int*)p) = object.i;
            if (type & IsLong) *((long*)p) = object.l;
            if (type & IsFloat) *((float*)p) = object.f;
            if (type & IsDouble) *((double*)p) = object.d;
        }
    }
}
```
Implementation Issues

```c
int getValue(char *sens, mytypes_t *obj) {
    static int oldcpuvalue, newcpuvalue;

    // first sensor: cpu
    newcpuvalue = cpu();
    printf("getValue: new==%d, old==%d\n", newcpuvalue, oldcpuvalue);
    if (newcpuvalue != oldcpuvalue) {
        obj->i = oldcpuvalue = newcpuvalue;
        strcpy(sens, "cpu");
        return 0;
    }

    // second sensor ...
    // etc

    // no value available: return -1
    return -1;
}
```