C/S vs. Centralized

Schneberger's study was a survey of experience programmers who have programmed both kinds of systems, to capture their impressions

programmer: 38%
progrmr team leader: 24%
systems analyst: 16%
project manager: 12%
other: 95

Average number of years
writing programs professionally: 11.03
writing only new software: 4.76
only maintaining software: 4.32
W & M SW for centralized arch.: 7.32
W & M SW for distributed arch.: 2.69

Percentage of work devoted to maintenance: 37%

Percentage of programmers in organization
doing some maintenance: 57%

Programmers at various branches of IBM and AT&T

From the questionnaires concluded:

Hypothesis 1: SW maintenance is more difficult the more difficult the computing environment
SUPPORTED

Hypothesis 2: The variety of computing components has a greater effect on SW maintenance difficulty than the number of computing components
OPPOSITE SUPPORTED

Hypothesis 3: The variety of computing interaction has a greater effect on SW maintenance difficulty than the variety of computing components
SUPPORTED

Hypothesis 4: The rate of technological change has a greater effect on SW maintenance difficulty than either the variety of computing components or the variety of interactions
SUPPORTED

Conclusions

Software for distributed systems seems harder to maintain than for centralized systems.

As for whether they were using libraries, the study does not say at all. However, presumably they were. Also it seems likely that there were extensive libraries for both kinds so that the presence of libraries ends up not being relevant.

More on N-version programming:

First a possible valid use, pointed out by Guy Levanon:

I think that, as you say, in most cases it's useless, but in some cases it's not.

If you give very specific requirments about the output but none about the algorithm, then N teams would develop N programs each with exactly the same output format but with different algorithm.

The result is that every algorithm is useful for a different input.

If the main problem is the algorithm used and not just writing bug free code, it may be useful to to use the average output of the N programs or to have N votes on a single result.

For example :

A basic operation in image processing is finding the different objects and identifing their borders in the image (segmentation). Many algorithms have been invented, and each is good for a different image type (picture from a camera of different views, paintings, pictures taken from airplanes, satelites, etc.). If the output requirments is the same without specifing the algorithm, then different algorithms will certainly be invented, and possibly each is good for a different type of image.

About the cost: althuough N version is a little less expensive than N times the cost of a single version, it may be worth it. It's worth to spend five times the normal cost to develop a better object recognition program or better weather forecasting program.

This principle is the basis of new research areas such as genetic algorithms. I know it's not a software engineering point of view, but it's not corrrect in my opinion to rule it out on the spot.

But I just saw an interesting paper by Les Hatton, ‘‘N-Version Design Versus One Good Version’’ in IEEE SW, November/December, 1997

(I remember discussing it with him over beer in Berlin in March 1996 at the Hard Rock Cafe’!)

Les is aware of the study by Knight and Leveson that shows that the assumption of independence does not hold among N independently developed programs for the same requirement. There are too many common-mode failures.

But Les still asks the fundamental question:

Is it more cost effective to develop one exceptionally good program or N less good ones, and which is more likely to lead to the more reliable system?

Les takes the Knight and Leveson data, uses them in different calculations and and finds that

While the typical N-version voting program is no where as reliable as one would expect given the independence assumption, still the N-version voting program consisting of high quality individual programs is more reliable than any one and is significantly so.

He determined from the data that the average improvement in reliability in a majority-voted 3-version program is by a factor of 5 to 9.

If this is so, then it IS cost effective to make 3 versions to get a 5 to 9 fold improvement in reliability, which is very hard to do by other means that are focused on single versions.

More on the CMM

There are some actual data finally on the effectivenss of the CMM.

‘‘Results of Applying the Personal Software Process’’ by Pat Ferguson, Watts Humphrey (prime mover of CMM), Soheil Khajenoori, Susan Macke, and Annette Matvya, in IEEE Computer, May 1997.

The SEI recommends that each programmer in an organization seeking to improve its CMM rating learn the PSP.

The main elements of the PSP is that each person begins to keep a notebook of how he/she spends his/her time, detailed down to the 15 minutes.

Once in the habit of doing this, he/she begins to be more disciplined in his/her programming.

The paper describes some empirical studies of PSP trained programmers.

SEI’s data on 104 engineers show that, on average, PSP training

- reduces size estimating errors by 25.8%
- reduces time estimating errors by 40%
- increases average number of lines of code per hour by 20.8%
- reduces portion of engineer’s development time spent compiling by 81.7%
- reduces testing time by 43.3%
- reduces total defects by 59.8%
- reduces test defects by 73.2%

Not bad!