Stress test

Objective:

- **Show unacceptable problems with high parallel load.**
- **Crash, wrong processing, slow processing.**

Test Procedure:

- **Run test cases with maximum number of parallel users.**
- **Same input for every user.**
- **Usage of same data element for every user.**
- **Test over short time.**

Result wanted:

- **No problems**
- **Reasonable response times.**
- **Reasonable error messages.**
- **Reasonable emergency service (graceful degradation).**
- **Scalability works.**

Considerations:

- **Data generation may need analysis of a usage profile and may not be trivial.**
- **Copy of production data or random generation.**
- **Use data generation or extraction tools.**
- **Data variation is important!**
- **Memory fragmentation important!**
- **Data should follow usage profile.**
- **Try to generate overload, even if outside of specification!**

Stress test is a test to check if there are problems with maximum parallel load on a system. This test is not easy, and because of this, many systems have not been tested for stress. Systems may crash, loose or corrupt data or get too slow. Stress test should be run with maximum loads and even overload until the system crashes or its performance degrades, in order to measure bottlenecks. The test also measures scalability

General candidates for stress test are situations where a maximum number of input channels are active at the same time.

Typical problems are time-out, full or nearly full buffers as well as counters which get out of bounds or overflow, or situations where the system administrates itself ”to death”.

A special variant is a test of especially little or even no traffic on the system over prolonged periods of time in order to see if the system ”falls asleep forever” or closes connections where it should not do so.

One last variant is to test or measure how much processing, network or memory capacity is needed by a program. This is important to measure if the program shares a processor with other programs. The sum of the maximal necessary resources must not be above the available resources.

Examples:

Online system: Input as fast as possible, but not above what can happen in use, from a maximum number of parallel channels. Include use of macros or shortcuts, as used by advanced users.

Data base system: Maximum number of logged on users. Maximum numbers of parallel transactions, even against the same table or data element. (Prototype situation: Everyone wants a ticket to the same concert.) A large number of complicated transactions or a large number of parallel wrong actions or cancellations (for example alarms if a network cable fails).

File exchange: This test is only interesting if files are sent in parallel over a network. Test sending from every channel at the same time and check for collisions, timeouts or incomplete transfer. Too long waiting times may also be an interesting result.

Disk space / file system: Stress test can be combined with volume test for testing really extreme situations. An example is low memory on all platforms at the same time as parallel load increases.

General checkpoints

- Warnings and error messages, do they come at all and are they helpful and understandable?
- Are data lost or corrupted?
- Is the system too slow?
- Do timeouts happen?
- Does the system crash?
- Are really ALL data processed and stored? Even the end of files, messages or data elements?
- Are data overwritten without warning?
- Does the system prioritize in a reasonable way at overload situations? Are prioritized services still available?

How to find test conditions

Test conditions are any situations that may lead to problems.

The description here is for administrative systems like database systems.

Discuss the following situations:

- Maximum number of clients at the same time
- A large number of advanced users (use of macros, shortcuts, fast input, ...)
- Login at the same time
- Same action at the same time
- Access to same data element at the same time
- Maximum number of users logged in, but not so many active ones
- Maximum number of users logged in and active
- Wrong use from many clients
- Hardware problems like database server going out of service or some of the servers in a server park crashing.
- Network connection through rerouting or backup solutions
- A large number of complicated actions at the same time (for example searching and sorting through many tables)

Identify points of time where the use of the application is special. Examples are:

- Friday afternoon (reports should be ready)
- End of periods (quarter, half year, year)
- Days with especially much traffic (last weekday before a holiday or tax deadline)
- First day after a campaign
- Customer panic
- Last day before and first day after rules or taxes are changed
- Situation right after a crash (everyone logs on)
- Large batch traffic at the same time as large online traffic.

Extra for web applications:

- Robustness against distributed denial of service attacks
- Situations where the internet connection of the server is upgraded (capacity increase of network)

Analysis of test result:

- Does the service crash?
- Unacceptable response times?
- Data lost?
- Data corrupted?
- System crash?
- Is it possible to come round security control in such situations?

Find places where data is stored temporarily. Find the functions storing there. Find functions taking away these data (delete). Make scenarios going through all of these functions. Find if such storage can become full or overflow. Does it happen that temporary data, which should be deleted, are actually not deleted and thus deplete resources?

Can we cut out stress test?

Yes, if stress is either no issue (only one user) or stress testing has been done by someone else before and the result is documented and good enough.
For example of the supplier has done such a test before on a configuration which is equal to or less powerful than the platform to be used.

Be careful when integrating several independent systems on the same platform: It must be guaranteed that every system gets the resources it needs according to specification. If this cannot be guaranteed it may be best to run a stress test on all integrated systems. This includes maximum load on all systems, i.e. the whole architecture.

Checklist:

If at least one question is answered NO, then stress test should be done.

- Has a stress test been executed before on the platform to be used or a less powerful platform and can the result be checked?
- Can it be guaranteed that the system will always get the resources it needs (are specified)?
- Can the guarantee be given even if several systems share the platform?
- Is it guaranteed that the load will not increase beyond the specified maximum load?
- Is there low risk if the system gets in trouble at increasing load?

Building and running this test

- Test only the most important functions! Stress test is expensive and should be related to applications risk.
- Test cases and user input data from normal functional test cases.
- Possible to randomize in order to get more data.
- Production data may be copied, but then need anonymizing etc.
- Test environment should be same as expected use environment.
- Up to ten parallel users can be replaced by human testers.
- Simulate large numbers of parallel users.
- Even blend of automatic driven test and manual test possible.
- First low stress, then more and more.
- Add load until problems occur, then analyze using measurements.
- Be aware that test tools may “steal” machine resources (probe effect).

Structure for automated test

Test clients can be real users or test robots or a blend of these. Test robots may replace any part of the architecture, i.e. they may simulate clients, network traffic or even be located on some remote server, if the objective is to test this only. It is possible to simulate many clients on one platform. Then a "test controller" controls test clients. They behave like users (should be programmed to do so), executing each one's own test script against the application under test. They log the system responses either locally or send the answer to the test controller for archiving and evaluation.

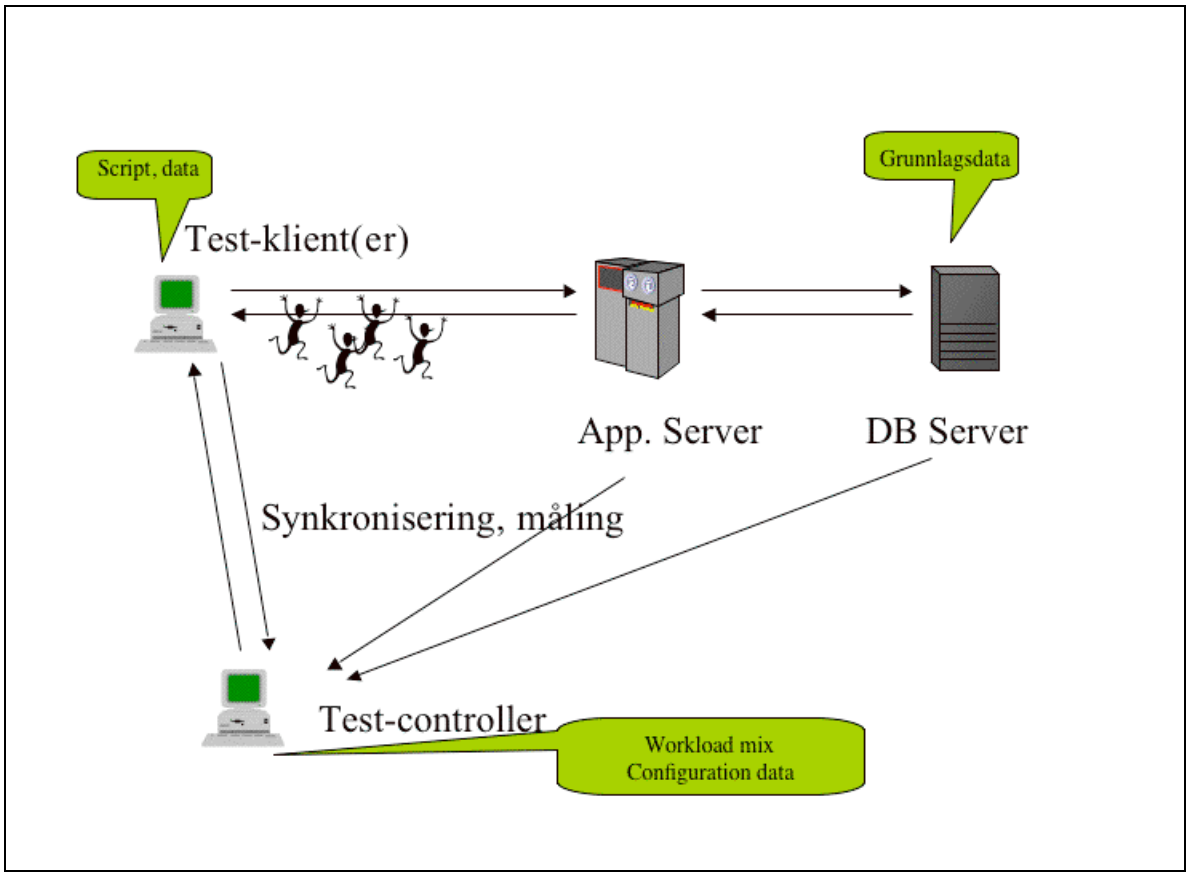
In principle, the test clients behave like Internet clients in the well known distributed denial of service attack (DDOS). This means many test clients wait for the trigger signal for them to become active and then send transactions or requests to the system under test. They may then send requests with different speed, even maximum speed.

Test clients can be programmed with standard script languages like perl, Ruby or as object oriented programs which instantiate instances of users (Java, C++, C#, Ada). Advanced tools can be found by searching the Internet for "load test tools" m "stress test tools" or "performance test tools" as well as by studying www.testinfaqs.org.

In order to aid debugging, monitors should be activated for every interesting resource, everything that may become a bottleneck. These are, amongst others, network connections, every server, operating system, CPU, coprocessors, database, disk etc. Such monitors are partially included in operating or database systems, partially platform dependent and must be acquired.

The probe effect: Both simulation of users by test driver tools (test robots) and all the monitoring may steal resources. This means the system under test has fewer resources than in real use and may thus run slower or even get problems. This is called the probe effect. This is a problem if such tools take a lot of resources away. However, failures are to the safe side, as the final system will have more resources available than under test.

Below is a figure showing the structure of such a test. Sorry for Norwegian texts.


Script, data
Test-klient(er)
Grunnlagsdata
App. Server
DB Server
Synkronisering, måling
Test-controller
Workload mix
Configuration data