Performance targets are usually given at the end user or overall system level, such as an end to end response time target or an overall throughput target. After the architecture and design phase of any IT system, and before commencing development, one needs to be clear as to what targets to provide to developers of the IT system or application. However, other than functional specifications, nothing much is provided in terms of targets and this leads to significant budget overruns later on or even projects landing up in failures. This document provides an understanding on two important aspects for the developers. First, how to arrive at a performance target and second, how to bridge the gap between production and development.
Performance Targets for Developers

Rajesh.Mansharamani@swperfengg.com

November 2011

The white paper on ‘Performance Requirements Gathering and Analysis’ introduced us to performance requirements gathering and analysis from the point of view of the business. These requirements serve as inputs to a software implementation project. Architects design the architecture of a software system and plan for suitable hardware capacity to meet the business requirements provided by a customer. It is typical for large projects to have a small core team of solution architects who have significant experience in the software industry. This contrasts dramatically with the development phase of the software implementation project.

During software development of any large project there are a large number of developers, often in the range of 50 to 100. To manage project costs, it is typical for a large percentage of the developers to be junior. One cannot expect the junior developers to match the experience levels of architects. Building and analysing performance models is simply out of the question, as far as developers are concerned.

This leads to a fundamental problem with every large project. On paper, the architecture and design are critically reviewed and given a green signal. However, many projects run in to disaster since the output of development is nowhere near where the paper plan said it would be. To compound this problem, there is poor root cause analysis of the disaster due to lack of analytic abilities in developers.

Mature organisations today rely a lot on performance testing to evaluate software and system performance well before it hits production. However, at this stage it is very late to make changes in architecture and design, and many projects get delayed or are wound up once the test results are available. This happens despite the elaborate software development process and its reliance on checklists for code optimization.

Checklists are simply a means to achieve an end. So if the end is not clear, what can be expected of the means? As of today, there is no well established means in the industry to set performance targets for developers. That is what we address in the first part of this document. Thereafter, we move on explicitly laying out the gaps between development and production environments, which need to be understood very well by all developers. Unfortunately, these gaps are never made explicit in today’s software development projects; these issues can potentially cause severe problems while managing production systems in large and complex projects. We end this chapter by introducing performance emulation which is a means to bridge the gap between development and production environments.

For the sake of completeness we include the theoretical foundations that may already be available in the other white papers published in this site.
1. Theoretical Background

In this section we derive two fundamental laws. First Little’s Law that provides a relationship between number of users, response time and throughput. Second, bottleneck law that derives the maximum possible throughput of any system in terms of service demand (which will be clarified below).

1.1 Little’s Law

Little’s Law is a well known theorem from operations research, which was published in 1961 by J.D. Little. We use a simplified version of it for closed systems (having a fixed number of users). Consider a system, as shown in Figure 1, with $N$ (business) users, with an average response time of $R$ seconds per business transaction, and having a throughput of $X$ business transactions per second. (We could might as well as use business interaction or web page per request.) Let us assume that the average think time per user is $Z$ seconds, per business transaction. By think time we mean the time that a user spends in data entry, or in viewing results, or in doing anything other than waiting for a response.

![Figure 1: Closed System](image.png)

Now, as shown in Figure 2, the average cycle time for one user is $C = R + Z$ and therefore the throughput per user is $1/(R+Z)$. For the overall system we get:

$$Little\, 's\, Law\, for\, Closed\, Systems:\ R = X \times C \quad = X \times (R + Z)$$

The only assumption behind this law is that the system is work conserving, in that no work is generated within the system and no work is destroyed by the system.

For example, if a system has 3,000 concurrent users and a throughput target of 100 transactions per second, the average cycle time per transaction is $3,000 / 100 = 30$ seconds. If the think time per transaction is 25 seconds, then the average system response time target is $30 - 25 = 5$ seconds.
1.2 Bottleneck Law

Before we get on to this law let us elaborate on some important terms, namely, service time, visit count, and demand.

Service time is the time spent by a resource in servicing a single request. For example, a single banking transaction makes 5 requests to a web server with an average of 5ms CPU time per request, 2 requests to an application server with an average of 10ms CPU time per request, 4 requests to a database server with an average of 20ms CPU time per request, and 10 requests to the disk subsystem with an average of 5ms disk service time per request. Note that the service times in this example are time spent in servicing the request, and they do not including queuing time or wait time at the resource, which forms part of response time. In other words, service time at a resource can be thought of as response time at the resource under idle conditions.

We use the symbol $S$ to denote average service time.

In the previous example we saw that a single transaction makes multiple visits to sub-systems and resources. The average number of visits to a resource is called the visit count of that entity at the resource. Note that visit count by definition is an average. Also note that visit count is a relative number. In the example above, one banking transaction makes 4 requests to the database server and 10 requests to the disk subsystem. Thus the visit count is 4 at the database server and 10 at the disk subsystem. This is relative to the banking transaction. At the disk subsystem the visit count relative to the database is 2.5 (10/4). Visit count can also be a fraction which is less than one. In the example above, if we have 8 CPUs at the database server, then the visit count per CPU is 4/8 = 0.5.

We use the symbol $V$ to denote visit count.

Whether we make 4 requests to the database server with service time 20ms per request, or 1 request with service time 80ms, the total service demand at the database server remains the same, that is, 80ms. Thus the average demand at a resource is the product of average service time at that resource and the visit count at that resource.

The symbol $D$ is used to denote average demand. Thus at each resource $i$ in the system the average demand is:

$$D_i = V_i S_i$$

Now let us get down to the Bottleneck Law. More specifically, we are looking at an upper bound on throughput or the maximum throughput that is achievable. Using that we can also derive a lower bound on average response time thanks to Little’s Law.

Consider an end-user system as illustrated in Figure 2. An entity requesting services of the system visits several resources, with a certain visit count and average service time. The circles in the system denote resources, and the tuples shown next to the circles specify the visit count and average service time at the resources.
As defined in above, the average demand at a resource is the product of the visit count and the service time. For the purpose of our analysis of demands, we can equate Figure 2 to Figure 3, which shows the system as a pipeline of resources each having service time equal to demand. In other words, instead of specifying that a resource $i$ is visited $V_i$ times with an average service time of $S_i$, we specify that the resource is visited once with average demand of $D_i$. For the purpose of the bounds derived in this section, this translation works appropriately.

If we consider any pipelined system such as the one in Figure 3, the maximum throughput of the system cannot exceed the throughput at the slowest stage of the pipeline. In the example in Figure 3, the maximum throughput of the system is $1/5$.

Let the maximum average demand in the system, across all resources $i$, be denoted by $D_{\text{max}}$:

$$D_{\text{max}} = \max_i \{D_i\}$$

We therefore have the upper bound for system throughput as:
\[ X \leq \frac{1}{D_{\text{max}}} \]

This is what we mean by the **Bottleneck Law**, i.e., the bottleneck resource determines what the maximum overall system throughput will be.

The upper bound holds, regardless of the system workload. When the system saturates this, the upper bound becomes an equality. By definition $D_{\text{max}}$ depends on visit counts and service times. $D_{\text{max}}$ can be reduced by optimizing the software design and implementation to reduce service times, or by using faster CPUs or disks to reduce service times, or by increasing the number of resources at a service centre to reduce the visit count per resource, or by changing the architecture of the system to reduce visit counts. For example, if database visit counts are high, one can either increase the number of CPUs or disks, or introduce caching at the application server in order to reduce the visit counts.

From Little’s Law we get:

\[ R = \frac{N}{X} - Z \]

We now apply the upper bound on throughput to get a lower bound on average response time.

\[ R \geq ND_{\text{max}} - Z \]

These bounds on throughput and average response time, become equalities upon system saturation (unless the system is not work conserving and thrashes after a certain load).

Now that we are done with the requisite theoretical background, let us get down to the main task of deriving targets for developers from end user performance requirements.

## 2. Deriving Performance Targets for Developers

A developer, typically, writes code for one unit at a time, where a unit could be a single class in an object oriented programming language like C++ or Java, or a unit could be a single method in that class or a function in a language like C or PL/SQL. In state of the art software development, functional specifications are provided for each unit. The units are developed in any standard language such as Java, C++, C#, and/or SQL. Thereafter, the units are tested against the functional specifications in what is called unit testing. Throughout this process, performance is missing.

Performance awareness manifests itself only during performance testing, which occurs after all units have been assembled and the software system runs successfully end-to-end. That is, after integration testing. This leads to several challenges in meeting end-to-end performance goals:
• The end-to-end performance requirement could fail due to bottlenecks in multiple units. This leads to a blame game, especially when ownership of units is spread across multiple vendors. To ease out this problem there is a need for significant investment in performance monitoring and diagnostic strategies and tools.

• The cycle of test, monitor, identify bottlenecks, fix bottlenecks and retest creates a long cycle. In large systems, tests run for hours and fixes come after a few days for minor to moderate changes. At the time of fixing a bottleneck there is no performance specification for that bottleneck unit. Hence the only way to assess against end-to-end performance targets is to run end-to-end performance tests.

Here’s an analogy to illustrate the problem. Say you have to travel from Mumbai to Pune by car without a speedometer. You know that you need to reach Pune in 3 hours. Your car has had problems in speeding up and to address this, the service agency tries out a quick fix. However they are not sure that this fix will help you achieve your goal. So the agency asks you to travel to Pune and return to Mumbai to determine if you can do it in 3 hours when there is no traffic. In short, you have to run the end-to-end performance test to assess performance against the end-to-end target.

• Often bottlenecks are because of poor design and sometimes because of poor architecture. Any change in design and architecture causes a significant impact to the rest of the software system. This results in a lot of re-engineering, re-testing for functionality, and then testing for performance. This is because there is no performance target available at a component level, or at a unit level. There have been several instances of projects being delayed by months because of such design and architecture changes determined late in the software development cycle.

Once a target is made available to a developer in terms of the time that needs to be spent in a software development unit, the developer will automatically search for the means to meet the target. Meeting the target will cause a very high success rate during performance testing. Not meeting the target will cause the right escalations to be made very early in development. The escalation could result in a change in design or architecture, an upgrade of capacity, a better way to optimise code, or even a relaxation of targets.

Now let us get in to a formal analysis of target setting. At a given tier, we would like to know the service time. (Recall service time is the time spent in servicing a single user at a given resource, under no contention for the resource, that is, when the resource is idle.)

Consider a system comprising of $N$ users, with an average response time requirement of $R$, average think time per user of $Z$. Then as seen from Little’s Law in we have system throughput as:

$$X = \frac{N}{R + Z}$$
From Section 1.2, we saw that

$$X \leq \frac{1}{D_{\text{max}}}$$

which means that $D_{\text{max}} \leq 1/X$

where $D_{\text{max}}$ is the maximum of the average demand across all resources, and $X$ is the overall system throughput (say in business transactions per second).

At a given resource tier, the average demand per resource is the visit count multiplied by the average service time. Our methodology to determine average service time is given in Table 1:

**Table 1: Methodology to determine service time targets for developers**

1. From the business requirements determine overall system throughput (use Little’s Law if the throughput is not explicitly specified)
2. From the throughput requirement determine $D_{\text{max}}$, which is the maximum demand across all tiers
3. From the architecture of the system determine the visit count per resource tier $i$
4. Determine the visit count per resource at tier $i$, $V_i$, by dividing the visit count at that tier by the number of resources at the tier
5. We must have $V_i S_i \leq D_{\text{max}}$, which means that $S_i \leq D_{\text{max}} / V_i$

Let us illustrate this methodology with an example. Consider Figure 4 which shows a closed system with 3,000 users, an average response time of 2 seconds, and an average think time of 28 seconds.

**Figure 4: Example System for Service Time Target Computation**

In this system a single business transaction results in 1 HTTP request and 5 database calls. Capacity planning has been provided such that the application server will have 12 CPUs and the database server 20 CPUs. In this example we will apply the methodology from Table 1 to the CPU service time targets. The same methodology applies to disk service times as well.

Applying the methodology to the application tier we get:

1. Throughput $X = 3000 / (2 + 28) = 100$ requests/sec
2. $D_{\text{max}} \leq 1/X = 10\text{ms}$
3. Visit count at application tier = 1
4. Visit count per application tier CPU = 1/12
5. Average service time per application tier call ≤ 10 / (1/12) = 120ms

Therefore, each servlet call must consume at most 120ms of CPU on the average. To measure this time during development one can use profilers available in the market, such as CA Wily, or run a load test (with a small number of users), determine the utilization U, and compute average service time \( S = U / X \). (See Exercise at the end of this document for this Utilization Law.)

Applying the methodology to the database server in Figure we get:

1. Throughput \( X = 3,000 / (2 + 28) = 100 \) requests/sec
2. \( D_{max} \leq 1/X = 10\)ms
3. Visit count at application tier = 5
4. Visit count per application tier CPU = 5/20
5. Average service time per application tier call ≤ 10 / (5/20) = 40ms

Therefore, each database call must consume at most 40ms of CPU on the average. To measure this time during development, tools such as TKPROF (in Oracle) or SQLProfiler in MS SQL.

Note that the service time targets are determined before the start of development. They can be easily tracked using off-the-shelf tools. If the targets get violated during development, the appropriate rectification mechanism can be taken up. This could be, for example, optimizing the code, modifying the design and/or architecture, or revising the capacity or the workload.

A word of caution needs to be mentioned here. We need to ensure that the overall response time is bounded. For example, if we need a throughput of 100 requests/sec and this means a maximum average demand of 10ms, consider the situation where there are 300 resources to traverse through for a business transaction response, and each has a demand of 10ms. Here the overall average response time has to be at least 300 x 10ms = 3sec which would be in excess of 2 seconds. Note this is a rare case where a business transaction needs to traverse 300 resources. To complete the transaction, it would require that the sum of the average demands across all tiers is less than the total average response time, that is, \( \sum D_i \leq R \).

In the above discussion, we have skipped an important aspect that a development environment is often of a much smaller size than a production environment. Thus functional tests will reproduce properly in production, performance tests will not. More specifically, while the targets for service time have been set in accordance with the capacity planning of the system, how do we account for latencies in wide area networks (this is not a capacity challenge but a physical distance challenge). Or for that matter, how do we account for a development database being populated with data from 1,000 customers while a production database may have data from 1 million customers. This is an important area of coverage and we dedicate the next two sections to it.
3. Gaps Between Development and Production Environments

Table 2 provides a comparison between development and production environments as we see them today.

**Table 2: Development versus Production Environments – A Performance Perspective**

<table>
<thead>
<tr>
<th></th>
<th>Development Environment</th>
<th>Production Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Users</strong></td>
<td>Developers</td>
<td>End users of the business</td>
</tr>
<tr>
<td><strong>Number of Concurrent Users</strong></td>
<td>Typically, tens of users</td>
<td>Tens to thousands of users</td>
</tr>
<tr>
<td><strong>User Access</strong></td>
<td>Over LAN</td>
<td>Over WAN, spread across multiple geographies</td>
</tr>
<tr>
<td><strong>Database Volumes</strong></td>
<td>Typically enough to do functional testing, of the order of ten to thousand records</td>
<td>Millions to billions of records for large databases</td>
</tr>
<tr>
<td><strong>External Interfaces</strong></td>
<td>Typically absent</td>
<td>Typically, tens of interfaces to external applications both within and outside the enterprise</td>
</tr>
<tr>
<td><strong>Server Size</strong></td>
<td>Often 2 to 4 CPUs</td>
<td>Large systems often go up to 32 or 64 CPUs per server</td>
</tr>
<tr>
<td><strong>Provision for High Availability</strong></td>
<td>Usually absent</td>
<td>Usually present, either in active active mode or active passive mode</td>
</tr>
</tbody>
</table>

As seen from Table 2, there is a significant gap between development and production environments. In fact, several of these gaps are also present in test environments versus production environments. Given these gaps, major performance issues often go undetected in development environments. If the test cycle does not address the gaps, these issues go forward into production and will most certainly lead to crises.

For mission critical environments there is a lot of emphasis on the performance test cycle. Performance testing provides for simulated generation of user load on production like servers and storage, and sometimes also reproduces database volumes. The gap between LAN and WAN is significant and is often skipped during tests and quite often external interfaces are also ignored during tests.
As discussed earlier, if major issues surface during testing it is very late in the project to make design and architecture changes and again very late in the project to make major code changes. Often project schedules slip significantly because users do not accept the performance that they see in production. It would have been ideal to ‘see’ the production environment during development.

How we bridge these gaps between development and production is covered in the next section.

4. Performance Emulation

By emulation we mean that we would like to see a ‘mirror’ of production as far as the end user is concerned without investing in a production environment. Note that emulation is different from simulation in that it caters to the net effect perceived by the end user. Simulation on the other hand is used to model a system for analysis purposes only. For example, you have a 10 Gbps LAN and you wish to mirror a 2 Mbps WAN on to it. Then the emulation needs to cater to bandwidth, latency, and other network characteristics such as availability, and be such that the end user experiences a 2Mbps WAN on the LAN.

An emulator is a device that provides for emulation. In the absence of emulators, software development rarely caters to production environments. Keep in mind that it is only wishful thinking that nothing will go wrong in production. In mission critical or quality critical situations, experts are positioned in the project to ensure that the software released will cater to production-like scenarios. Often experts either talk in a language that junior developers cannot understand or the experts do not get the requisite amount of data to take decisions during development.

For an emulator to be successful we need it to address to the following requirements:

- Reproducibility
- Flexibility
- Transparency, and
- Simplicity

By reproducibility we mean that it needs to reproduce a production subsystem’s responsiveness, availability, and reliability as accurately as possible. By flexibility we mean that the emulator must provide controls to model the behaviour of a production environment to support ‘what if’ analysis. If the current capacity is not enough, then how much more would you need, can be answered if appropriate controls are provided. By transparency we mean that the emulator should integrate seamlessly with a development or a test environment, with the end user not noticing the difference and with developers and testers not having to undergo any process change or any skills upgrade. By simplicity we mean that the emulator should be easy to setup and use.

Once an emulator is available the controls are to be provided just to one or two key people in the project such as a project lead or a module lead. The leads can set the controls in the emulator and the environment should then perform in a manner similar to a production environment.
In this section we focus on network and database performance emulation. In the WAN emulation space there are three popular emulators: Shunra, NIST Net, and WANem. Shunra is a commercial tool whereas NistNet and WANem are open source.

In a tool like WANem, the user gets controls for setting bandwidth, latency, jitter, packet drop, and connection drop. WANem sits in a software CD or a VM, through which any desktop or laptop can be booted. Once the WANem desktop comes on the network, the client and server are made to talk through WANem by means of simple ‘add route’ commands on DOS or Linux or Unix. This command makes WANem like a gateway such that every packet from client to server and back is routed through WANem. Then WANem applies the controls specified by the user to every packet to make the net effect to the client and server as if it is a real WAN.

There is no product in the market to perform database volume emulation. The technique followed by IBM DB2 experts or Oracle experts is to manipulate database statistics to reflect production volumes and distributions. This influences the query optimiser to work as if the development or test database has production volumes, and it generates query plans that would show up in production. Developers then optimise their queries by looking at such query plans through ‘EXPLAIN PLAN’ commands.

This approach requires significant expertise by developers and does not provide any estimate of time that the query would take in production. The EXPLAIN PLAN commands do however provide query cost estimates that developers try to optimise. Moreover, the setting of statistics can be done up front by an experienced DBA. While the cost to response time relationship may not always be straightforward, at least for complex queries or badly tuned queries, cost is a meaningful indicator. Some IT service companies due build their internal tools to emulate response times in databases by using empiric models to estimate query execution time from costs.

**Useful Reading**

The material presented in this document is derived in part from some of the references below. Other references have been added as useful reading to supplement what is presented in this document.

Exercises

1. Little’s Law as we have seen it for closed systems states that \( N = X \cdot C \), where \( N \) is the number of users in the system, \( X \) the throughput and \( C \) the average cycle time. When we consider any open system where requests keep arriving and departing and the number of external users is not necessarily known then Little’s Law for that system is \( N = X \cdot R \), where \( N \) is the average number of users in the system, \( X \) the system throughput and \( R \) the average response time in the system. Now use this version of Little’s Law to prove the utilization law, as shown below.

Consider a subsystem with a number of identical resources, for example a CPU subsystem or a disk subsystem. Any request will undergo a service time and a wait time (if the resources are busy). Now focus only on the resource itself and not the queue for the resource. As shown in the figure below, draw the system boundary around the resources and apply Little’s Law. Now prove that utilization of the resource \( U = X \cdot S \), where \( X \) is the (sub)system throughput and \( S \) is the average service time at the resource. Note that for multiple resources at the subsystem \( U \) can be more than one. For example, if an 8 CPU server is 70% utilized then \( U \) is 5.6 (that is, \( 0.7 \times 8 \)).

2. A message queue needs to service 10,000 messages/sec on a 4 CPU server. Assume that all CPUs are equally utilised for messaging and the utilization needs to be within 70%. What should be the average service time per message?

3. A banking system with 5,000 concurrent users needs a 2 second average response time and has an 18 second average think time. Each banking
transaction makes 4 database calls to a 10 CPU database server. What should be the target for average service time per database call?

4. In Exercise 3 (above), one banking transaction makes 10 network calls. What is the service time target at the network? If each network call is 100 bytes, what is the minimum bandwidth required?

Answers to the exercises:

1. By Little’s Law average number in resource subsystem is throughput times average response time in subsystem. Average response time within the resource (outside of queueing) is average service time by definition. Average number in resource subsystem divided by number of resources is the percentage utilization of the subsystem. Hence $U = X/S$.

2. $U = X/S$. Since utilization is 70% we have $U = 0.7 \times 4 = 2.8$. Hence $S = 2.8/10,000 = 0.28$ms.

3. $X = 5000/(2+18) = 250$. Hence $D_{max} \leq 4$ms. Visit count per CPU = $4/10 = 0.4$. Hence $S \leq 4/0.4 = 10$ms.

4. $X = 250$, $D_{max} \leq 4$ms. $V = 10$. Hence $S \leq 4/10 = 0.4$ms. $S = \text{packetsize/bandwidth} = 0.1$KB/bandwidth. Hence bandwidth $\geq 0.1$KB/0.4ms = 250KB/sec.