Related Work

We can study the performance and operation of Java-based middleware systems using the RUBiS system. The authors of RUBiS showed that the mixture of requests—the workload—plays a significant role in determining system bottlenecks [1, 3].

Our approach differs from previous attempts to apply control theory to operating systems and three-tier systems, which have assumed that requests to the system are homogeneous (affect the system in the same way). The SWIFT system [6, 7] is a systematic approach to introducing input/output interfaces to operating system components, which matches well with the well-defined interfaces between middleware components. The ControlWare system [10] is a toolkit for automatically mapping QoS requirements into simple control loops in three-tier systems.

Considerable work has been applied to correlation analysis of Web services both in research literature and in industrial best practices. The SLIC project at HP Labs [4] attempts to identify which components are responsible for Web service violations of Service-Level Operations (SLOs) by using fine-grained monitoring and instrumentation. The Performance Management project at IBM has explored using control theory and statistical monitoring to detect and adapt to unexpected traffic surges [8, 5]. Techniques for visualizing structured data are described in [9].

Overload Avoidance in Self-Adaptive Web Services

Overload occurs when the load placed on a Web service exceeds its ability to serve requests. Flash traffic and sudden load spikes operate at timescales faster than operators can upgrade their systems. Web service operators can manage load in a number of ways. One way is to direct load to spare servers that can handle the surge. This technique is an example of load balancing. Complex Web services are often built in multiple layers of interconnecting components (see Figure 2). Applying a load-balancing strategy in this environment is non-trivial, since detailed instrumentation of the internal components is usually not available.

High-level overload mitigation strategies can be used, at least temporarily, during this time (such as HTTP 503 TOO BUSY responses). However, this adversely affects all traffic to the site, even when the bottleneck is driven by a small population of requests (about 15%, in our RUBiS emulation). This motivates the desire for a less disruptive, selective admission control.

In selective admission control, we first throttle back requests contributing to the overload, while leaving all other requests unaffected. In our implementation, the bottleneck was the database's CPU, and the two contributing requests involved searching for items. In general, it is quite difficult to determine the runtime connections between components in a distributed system. Often these are determined by the workload, and can change over time. In addition to the lack of visibility into these connections, it is non-trivial to map those connections from a request to a bottleneck(s). We seek to make use of measurement data in this process.

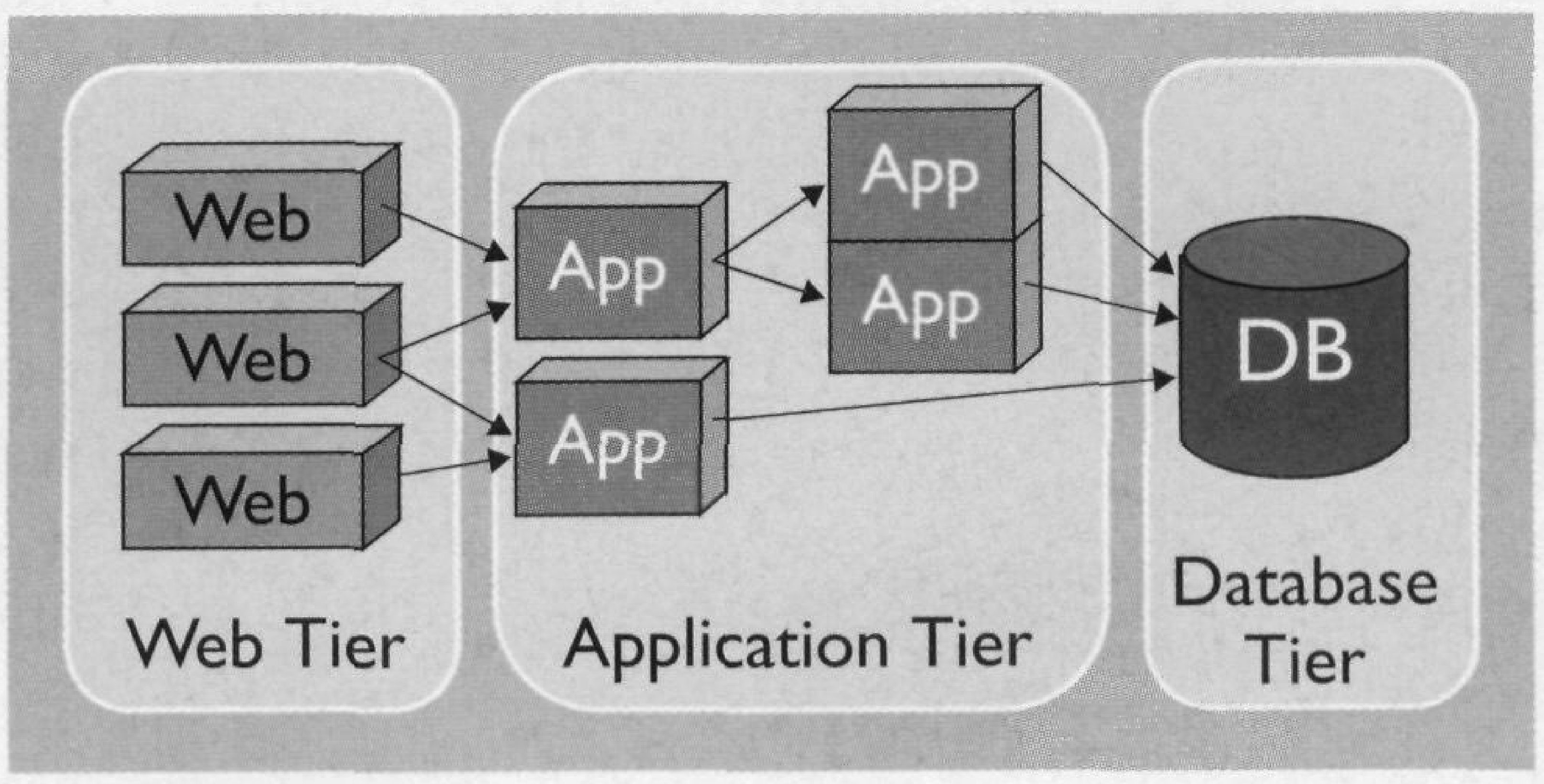

Figure 2. A complex Web service consisting of Web, application, and database components.

Problem Statement: Given a system bottleneck component C, identify those requests correlated with C. The data used for that purpose should be collected with minimal disruption to the system. Once identified, reduce the number of correlated requests until the system is no longer overloaded.

We now outline the four mechanisms of our approach in more detail.

Uncovering Request Effect through Correlations

When a request arrives at the Web server, it may invoke processing in one or more Java components in an application tier. In turn, these either access the database or return a result directly to the user. While logging and status information is available on each of the servers hosting these tasks, there are no good system tools for understanding crosscuts through the layers. Given the large number of possible crosscuts, we need a more sophisticated way of looking through the large amount of data collected at each point to discern correlations between components.

To find which requests are correlated with our bottleneck, we make use of the Apache Web logs collected from the Web tier and the CPU load average as