Давайте посмотрим, что нам предлагает современная индустрия.

С одной стороны есть Node.js и Aavatar.js. С другой стороны есть JavaEE сервера. Конечно, между ними есть серьезные технологические различия. Взять хотя бы языки программирования: в Node/Avatar.js это JavaScript, а в JavaEE это Java, но это не столь важно для настоящих профессионалов. Гораздо интереснее что там и там творится с моделями ввода-вывода. Как то так сложилось, что все привыкли считать, что JavaEE сервера ориентированы только на синхронную обработку запросов. Мол, у них есть стандартный пул потоков и поэтому временем ожидания ответа от третьего сервиса, базы данных и т.п. можно пренебречь. Опять же, как то так сложилось, что все привыкли считать, что Node/Avatar.js имеют асинхронную модель ввода-вывода и пренебрегают уже параллельной обработкой запросов. В последнем случае это так и есть и поэтому при промышленном использовании Node/Avatar.js люди вынуждены строить сервисы по схеме с многими экземплярами Node сервера, спрятанными за каким-то другим фасадом (nginx, apache). В рамках проекта Avatar.js, кстати, oracle-ом разрабатывается однопроцессная архитектура с несколькими экземплярами скриптового движка, обрабатывающими по собственному циклу обработки сообщений каждый в своем собственном потоке. А запросы принимаются балансировщиком нагрузки, распределяющим запросы для действительной обработки.

Теперь давайте рассмотрим поподробнее JavaEE сервера. Начиная с версии 3.0 спецификации Servlet API, в ней содержится раздел посвященный асинхронной обработке запросов. Только почему то это старательно замалчивается адептами Node-модели построения приложений. А там на самом деле все просто. У класса HttpServletRequest есть метод startAsync(), вызывая который можно «сказать» серверу, что обработка запроса на данном этапе завершена, но ответ в сеть посылать еще рано. Этот метод возвращает объект класса AsyncContext, с помощью которого можно управлять дальнейшей жизнью запроса и ответа на него. Например, там есть метод complete(), который завершает сеанс обработки запроса и посылает ответ в сеть. Очевидно, что эти две операции можно производить из разных потоков/контекстов и т.п. и ничто не мешает вызвать startAsync() непосредственно после посылки запроса к третьему сервису с помощью асинхронного клиента и в соответствующем callback-е такого клиента вызвать метод complete(). Сразу возникает мысль, что тогда потребуется асинхронный http-клиент и еще jdbc-клиент и так далее. И такие клиенты есть. Посмотрите хотя бы на <https://github.com/AsyncHttpClient/async-http-client> , <http://hc.apache.org/httpcomponents-asyncclient-4.0.x/>, <https://github.com/mheath/adbcj>.

Получается, что не все так очевидно, и там и там имеется асинхронная модель ввода-вывода и там и там есть возможность параллельной обработки запросов (в случае Node-модели много-процессный вариант с фасадами/балансировщиками). Вообще асинхронная модель ввода-вывода таит в себе несколько подводных камней. По крайней мере, я вижу два: неочевидный, тяжело читающийся код с множеством callback-ов и второй, следующий из первого, - запутывание прикладной логики. С другой стороны, очевидно, что просто так ждать у моря погоды (ответа от третьего сервиса) и ничего в это время не делать, - непозволительная роскошь. Но стоит ли говорить, что синхронная модель ввода-вывода, - это безусловное зло? У нее, как ни парадоксально, есть свои преимущества. Например, гладкий легко читаемый код, содержащий прикладную логику просто незаменим в учетных задачах (не полосой пропускания единой живут промышленные сервисы). Правда, тогда хочется чтобы в сервере бы пул потоков. Кроме того, в зависимости от характера сервиса, он может обслуживать запросы, обработка которых вовсе не связана с третьими сервисами и тогда модель ввода-вывода становится синхронной де-факто при любой технологии.

Асинхронная модель ввода-вывода хороша для приложений, для которых широкая полоса пропускания важнее прикладной логики. Это хорошо потому как потребность в доставке информации все время растёт.

Синхронная модель ввода-вывода хороша для приложений, для которых прикладная логика важнее полосы пропускания. Это хорошо потому как зарплату надо считать правильно и ничто другое не важно.

Конечно, каждый должен выбирать технологии для своего сервиса сам, но вот несколько критериев, по которым можно сравнить Node/Avatar.js и JavaEE.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Критерий/Технология | Асинхронный ввод-ыввод | Синхронный ввод-вывод | Языки программирования | Назначение сервиса |
| JavaEE | есть | есть | Java, JavaScript, Scala, jPython(Jyton), jRuby, Groovy | Анализ и доставка информации. |
| Node.js | есть | нет | JavaScript | Доставка информации |

Грубо, можно представить Node/Avatar.js и JavaEE как две противоположности (асинхронная модель ввода-вывода против многопоточности). Это часто встречающийся взгляд на эти две технологии. Однако нельзя ли взять от тех и других лучшее? Например в JavaScript-овых фреймворках, построенных на скриптовом движке Nashorn (Avatar.js, Platypus.js) вполне возможно написать код который будет исполняться в параллельных потоках, т.к. там можно пользоваться Java API [http://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/api.html](http://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/api.html.%20В%20Platypus.js). В Platypus.js <http://platypus-platform.org/> например уже реализован метод invokeBackground(), который вставлен в прототип функции и может вызвать ее в фоновом потоке.

В заключение хочется сказать, что в этой теме много «попсы» и маловато фактического материала. Хотелось бы побольше всестороннего анализа проблем, стоящих перед разработчиками сервисов, вместо голословных заявлений о неоспоримых преимуществах асинхронной модели ввода-вывода.